Aalto University - School of Science
Eurecom

Master's Programme in Security and Cloud Computing

Saba Feroz Memon

Resiliency in Kubernetes Federation Management

Master's Thesis
Espoo, August 25, 2021

Supervisors:	Professor Mario Di Francesco, Aalto University
	Professor Raja Appuswamy, EURECOM, France
Advisor:	Maël Kimmerlin, Ericsson, Finland

This thesis is a public document and does not contain any confidential information.

Aalto University - School of Science
Eurecom

ABSTRACT OF
MASTER'S THESIS

Master's Programme in Security and Cloud Computing

Author:	Saba Feroz Memon		
Title:	Resiliency in Kubernetes Federation Management		
Date:	August 25, 2021	**Pages:**	81
Major:	Security and Cloud Computing	**Code:**	SCI3113
Supervisors:	Professor Mario Di Francesco Professor Raja Appuswamy		
Advisor:	Maël Kimmerlin		

Over the years, cloud computing has gained immense popularity. The evolution of cloud computing has given rise to virtualization technologies, specifically, container-based virtualization. Kubernetes is the most widely used container orchestration tool, for deploying and managing containerized applications. Kubernetes provides a cluster with a control plane and several worker nodes. Due to the limitations of the single cluster architecture, multi-cluster deployment is being preferred as it improves scalability, isolation, and availability of applications. Kubernetes Clusters Federation provides a platform for managing application deployment on multiple clusters from a single management point called the manager. However, the manager is a single point of failure: if it goes down the application deployer cannot deploy applications on multiple clusters from a single point anymore. This thesis aims to address this issue by designing a Proof of Concept (POC) for achieving resiliency in the Kubernetes Federation management. The POC uses the active-passive high-availability approach for creating redundant managers. The managers use the Raft consensus algorithm for selecting a leader amongst themselves. The manager selected as the leader actively serves the application deployers and sends a copy of the configuration information of the deployed applications to the other managers. If the leader becomes unavailable, the Raft consensus algorithm selects another leader that starts from the last known state of the failed leader and continues serving the application deployers. The performance of the POC was evaluated by running a series of reliability experiments. The results of the tests were promising and showed that the proposed solution is highly reliable and feasible to implement.

Keywords:	resiliency, high availability, kubernetes federation, KubeFed
Language:	English

Acknowledgements

I would like to thank my primary supervisor, Professor Mario Di Francisco, from Aalto University for his valuable suggestions and review of my thesis. I also wish to express my deep gratitude to my thesis advisor Maël Kimmerlin for his guidance and support throughout the duration of my thesis and for the review of my report. I would also like to thank Jan Melen, my manager at Ericsson for offering me this thesis position and for his constant support throughout. Also, I would like to extend my thanks to Furkat Gofurov and Feruzjon Muyassarov, my colleagues at Ericsson for their valuable suggestions. I also wish to thank Professor Raja Appuswamy, my supervisor at Eurecom for his guidance and support. I also wish to extend my thanks to my SECCLO coordinators and Professor Tuomas Aura for their constant support and guidance throughout this Master's program. Finally, special thanks to my parents for their prayers and support. This thesis would not have been possible without their encouragement.

Espoo, August 25, 2021

Saba Feroz Memon

With the support of the Erasmus+ Programme of the European Union

Abbreviations and Acronyms

OS	Operating System
VM	Virtual Machine
CNCF	Cloud Native Computing Foundation
API	Application Programming Interface
CRDs	Custom Resource Definitions
CRs	Custom Resources
PCIDSS	Payment Card Industry Data Security Standard
NSM	Network Service Mesh
IPsec	Internet Protocol Security
CNI	Container Network Interface
NSE	Network Service Endpoints
NSC	Network Service Clients
NSR	Network Service Registry
NSmgr	Network Service Manager
MTTF	Mean Time To Failure
MTTR	Mean Time To Recovery
POC	Proof of Concept
RPC	Remote Procedure Call
Kind	Kubernetes in Docker
RBAC	Role-Based Access Control
VRRP	Virtual Router Redundancy Protocol

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, cloud computing has gained immense popularity and is now widely used around the world [56, 71]. Cloud computing helps reduce costs by providing on-demand IT products and services. Virtualization in cloud computing is the abstraction of resources that makes computing scalable, efficient, and economical. It provides a virtual image of resources that can be used on multiple machines simultaneously. Virtualization is done at the platform as well as at the Operating System (OS) level [55]. Platform virtualization virtualizes the hardware to run different Virtual Machines (VMs) on top of a single physical machine. Each VM runs its own OS and applications independent of the host machine and other VMs. OS virtualization relies on the host OS to run workloads in different computing environments called containers. Containers require only the application and the libraries they depend on to run; consequently, they are lightweight compared to virtual machines [67]. Container-based virtualization consumes fewer resources and makes the application deployment efficient and scalable [63]. It is important to have a container orchestration platform that manages the containers on multiple nodes in large-scale container-based systems. Kubernetes is the most popular orchestration platform that automates application deployment, resources management, scaling, and load balancing. It manages multi-tiered distributed applications as containers on a cluster of physical or virtual machines [69].

There are many limitations of deploying a single Kubernetes cluster, such as having a single point of failure, limited control over isolating multiple applications from each other causing them to impact each other, and requiring higher security measures for controlling the access of the users. Due to these limitations, organizations are moving towards deploying multiple Kubernetes clusters to fulfill their needs [12]. Deploying multiple Kubernetes clusters improves the scalability, isolation, and availability of the applications. How-

ever, in multi-cluster Kubernetes deployment applications are deployed and managed on each cluster separately [37], which is cumbersome. Kubernetes Clusters Federation makes multi-cluster application deployment easier by providing a framework to manage the application deployment on multiple clusters from a single management point namely, a manager. The application deployer communicates with the manager and applies the application configuration, which is propagated automatically to the multiple clusters as shown in Figure 1.1.

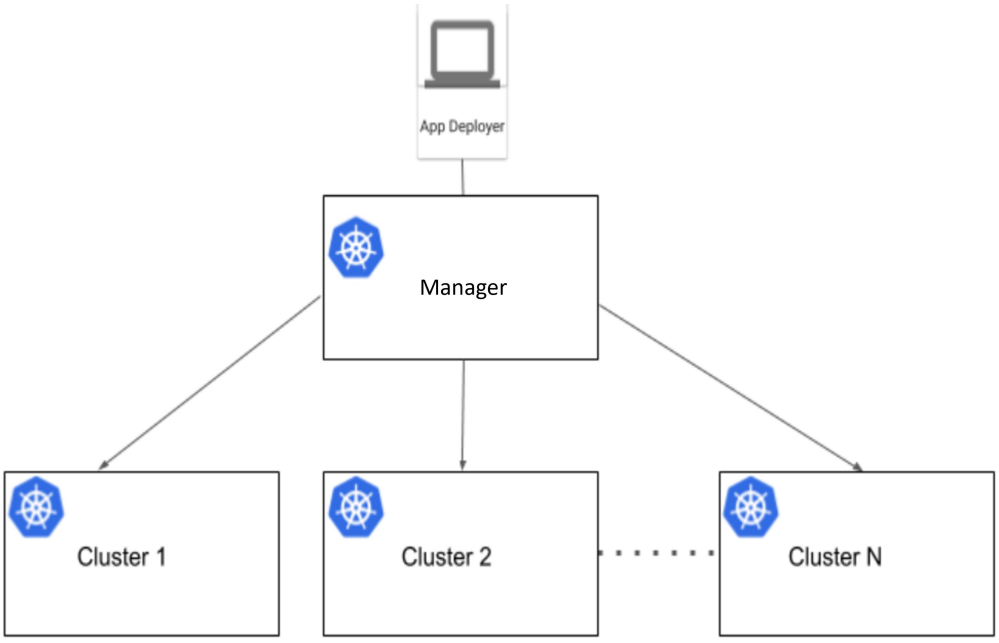

Figure 1.1: Multi-Cluster Federation

1.1 Problem Statement

In Kubernetes Clusters Federation, a central management point called manager manages the application deployment on multiple clusters and ensures that those clusters are in the desired state. If the manager becomes unavailable, the application workloads running on the clusters remain undisrupted. However, this scenario introduces a single point of failure. Having a single central management point prevents the application deployer from deploying or querying the federated applications if it becomes unavailable. Additionally, it is impossible to ensure that the federated applications on the clusters are in the desired state. Therefore, this thesis designs a Proof of

Concept (POC) to achieve resiliency in the Kubernetes Clusters Federation. The POC uses redundant managers to store the information about the deployed application. One manager actively serves as the leader and deploys the applications onto multiple clusters whereas the other managers store the configuration information of the deployed applications. If the active manager becomes unavailable, any one of the available managers can become active and continue the application deployment.

1.2 Structure

The rest of the thesis is structured as follows. Chapter 2 explains the main technologies this thesis is based upon with a focus on aspects related to resiliency. Chapter 3 describes different solutions for Kubernetes Clusters Federation along with their advantages and limitations. Chapter 4 describes resiliency and how it can be achieved using consensus protocols. It also explains and compares different consensus algorithms. Chapter 5 overviews the proposed solution and the POC implemented in this thesis. Chapter 6 explains the test environment and the experiments performed to evaluate the feasibility of the proposed solution. Chapter 7 concludes this thesis and provides directions for future work.

Chapter 2

Background

This section overviews the basic concepts behind the Kubernetes Federation with a focus on aspects related to resiliency. Specifically, its first section describes Kubernetes, its main objects, and the basic architecture. It then explains multi-cluster management in Kubernetes, different approaches for configuration of multiple clusters, and their benefits. The chapter also overviews Kubernetes Federation and discusses a reference Kubernetes Federation architecture and concludes with a summary.

2.1 Kubernetes

Kubernetes is a container orchestration system derived from Borg and Omega [51], Google's internal container-based cluster-management systems. Kubernetes is currently hosted and managed by the Cloud Native Computing Foundation (CNCF) [50]. A container orchestration tool ensures that containers scheduled to execute workloads run on either a physical or a virtual machine and replaces unhealthy containers. Kubernetes is the leading open-source platform that automates the deployment, management, and scaling of containerized applications [69]. It allows developers to focus on the business logic of applications without worrying about the complexity of implementing it.

When Kubernetes is deployed, it provides a cluster containing the control plane as well as the worker machines [49, 55]. The role of the control plane is to host the processes that maintain the desired state of the cluster. The worker machine is responsible for running the application workloads. Every Kubernetes cluster contains at least one worker machine also referred to as a node for running the containerized workloads. The control plane machine can also work as the worker machine and run the application workloads although

this is not generally recommended.

Kubernetes uses persistent entities called Kubernetes objects [31] to represent both the desired and the actual state of the cluster. They include, for instance, the currently running applications, the available resources, and the policies related to the application behavior. These objects are queried and manipulated through the Kubernetes Application Programming Interface (API) [22]. There are many Kubernetes objects. The following section overviews those relevant to this thesis.

2.1.1 Kubernetes Core Objects

- **Namespaces:** Namespaces [28] group Kubernetes objects by binding them to a name. Namespaces provide an abstraction through which the users create virtual sub-groups within the cluster. They allow different groups of users that belong to a specific team or project to share resources within a single cluster without interfering with each other's work. They also allow the users to enhance role-based access control, define resource quotas for different teams, and separate development, testing and deployment environments from each other [29]. Kubernetes provides the users with three namespaces, i.e., `default` for default resources, `kube-system` for Kubernetes components and `kube-public` for public resources. In addition, users can create their namespaces and deploy objects in those namespaces. When a Kubernetes cluster is deployed, all the deployments, services, and pods with no namespace are deployed in the `default` namespace.

- **Pod:** A pod [33] is the smallest deployable computing unit representing an instance of an application running within a cluster. It comprises one or more containers that share the same resources, such as storage and network [62, 69]. Each pod in a Kubernetes cluster is assigned a unique IP address shared by all the containers within the pods through which other pods communicate with it. Kubernetes supports two types of pods, i.e., *Service pods* and *Batch/Job* pods. Service pods run a workload in the background permanently whereas job pods execute specific jobs and terminate on job completion. A pod can be exposed to the outside world by configuring it to use the host network. This would enable the pod to access the network of the host node and be accessible on all the interfaces of the host machine. A pod can also be exposed to the outside world by using a Kubernetes Service as defined below.

- **Service:** A Kubernetes pod is ephemeral. It is created and destroyed

multiple times during its life cycle. Therefore, a Kubernetes service [35] acts as an abstraction for pods. Specifically, it exposes a group of pods by assigning them a name and a unique IP address allowing network access to those pods from within and outside the cluster. The service type defines the scope of the access to the pods [70]. There are three types of services.

  - **ClusterIP:** By default, the services assign a ClusterIP to the pods. The pods with a ClusterIP can only be accessed from within the cluster.
  - **NodePort:** The service is accessible from outside a cluster by assigning a static NodePort to it from the range 30,000-32,767. The NodePort serves as a proxy to access the pods selected by the service within the cluster.
  - **LoadBalancer:** provides access to the service from outside the cluster by enabling the load balancer functionality of a cloud provider. The load balancer then automatically redirects the requests from outside the cluster to a given service.

- **Deployment:** A Kubernetes deployment [26] acts as a controller for creating and managing the pods running the application workloads. It can be used for scaling up and down the pods, rolling out updates or rolling back the pods to an earlier state [25]. Users can define the desired state of the pods within the deployment; the deployment controller continuously monitors the state of the pods to ensure that they maintain the desired state.

- **ConfigMaps:** ConfigMaps [23] are used to store configuration information, such as connection strings, hostnames, credentials, or URLs about the objects as key-value pairs. In Kubernetes, ConfigMaps allow users to keep containerized applications portable, by decoupling the application logic from the environment-specific configuration. The information stored in a ConfigMap is not encrypted or kept secret therefore only non-confidential information should be stored in a ConfigMap.

- **Secrets:** Secrets [34] are used to store and manage a small amount of sensitive data, such as passwords, ssh keys, and OAuth tokens. A pod can use a secret by referencing it using three ways, i.e., as files in a mounted volume on a container, as container environment variables, and using kubelet when it pulls the images for the pods.

- **Custom Resource Definitions:** Kubernetes Custom Resource Definitions (CRDs) [24] are used to extend the Kubernetes API by adding Custom Resources (CRs). A Kubernetes Resource is a Kubernetes API endpoint that stores the collection of objects of a specific point. For instance, the deployment resources store a collection of the deployment objects. CRDs enable the users to define objects of a custom type and make them available as Kubernetes resources. The users can then create custom resources using then CRDs. These resources are then handled by Kubernetes and can then be used like all the core Kubernetes resources.

2.1.2 Kubernetes Architecture

Figure 2.1 shows the basic architecture of a Kubernetes cluster with one control plane and three worker nodes. The control plane manages the worker nodes.

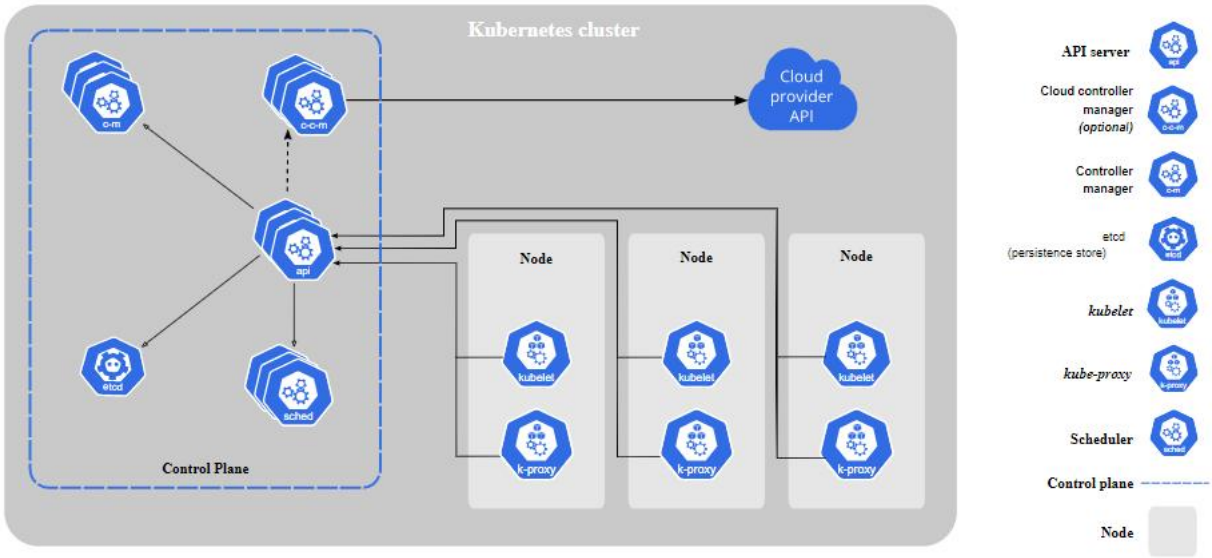

Figure 2.1: The Kubernetes Architecture [49]

The control plane contains the following components [49].

- **API Server**: The API server is the main component of the Kubernetes control plane. It exposes a RESTful API for querying and changing the state of the Kubernetes objects within the cluster over HTTP. The

API server acts as a front-end that is accessible by users and several other components from within and outside the Kubernetes cluster. The HTTP API can be used to modify objects to achieve the desired state of the cluster by using command-line tools, such as kubectl [36] and kubeadm, or directly through an HTTP client, such as wget [48] or curl [8].

- **Etcd:** Etcd is a highly available and consistent database that stores information about the cluster configuration, cluster state, and metadata. It is a distributed data store that stores the information as key-value pairs that are read and written by only the API server.
- **Kubernetes Controller Manager:** The controller manager is a daemon that implements the non-terminating control loops to maintain the desired state of the cluster. A control loop uses the API server to inspect the state of the pod and makes or requests changes.
- **Scheduler:** The scheduler assigns the most appropriate nodes to the newly-created pods by using different scheduling algorithms [54]. It ensures that the workload in a Kubernetes cluster is distributed according to the defined constraints, available resources, and other specifications. The scheduler can also remove a pod from a node to distribute the workload.

The Kubernetes worker node runs the pods that execute the application workloads. It contains the following components [49].

- **Kubelet:** is a node agent that runs on each of the Kubernetes nodes and registers the node to the API server. It receives the pod specifications from the API server and ensures that the containers are running inside pods according to the pod specification.
- **Kube-proxy:** is a network proxy that runs on each node within the cluster and is responsible for maintaining the network rules. These network rules enable internal and external network communication to the pods within a cluster.
- **Container Runtime:** is a software that pulls the images from public or private registries and then executes the containers using those images. A registry is a storage and distributed system for named images. Kubernetes supports multiple container runtimes, such as containerd, rkt, lxd CRI-O, and Docker. However, the most widely used container runtime is Docker.

2.2 Multi-Cluster Kubernetes

Multi-cluster Kubernetes refers to the strategy of deploying applications on more than one Kubernetes cluster. Although deploying applications on a single cluster is cost-effective and provides ease of management, such an approach has several limitations [3]. First, applications deployed on a single cluster have a single point of failure; any sort of fault in the cluster, such as component failure, configuration error, or outage in the infrastructure, can disrupt the entire workload. Second, multiple applications running on a single cluster share the same resources, such as hardware, network, and the operating system, and may interact with each other in undesired ways that are not secure. Moreover, these applications can utilize the resources designated to the other applications, thereby preventing them from running their workloads. Third, when all the applications in an organization are deployed on a single cluster all the employees get access to that cluster which imposes higher security risks in the cluster. Fourth, there is a theoretical limit up to which a single Kubernetes cluster can grow to manage the workload; 5,000 nodes, 150,000 pods, and 300,000 containers. Also, it is quite challenging to manage a single cluster when the pod size increases over 500 due to higher strain on the Kubernetes control plane.

As a result of these limitations, organizations have been moving rapidly towards deploying applications on multiple Kubernetes clusters. Section 2.2.1 overviews different approaches for multi-cluster Kubernetes.

2.2.1 Multi-Cluster Architecture

There are several approaches [3, 46] that can be employed to divide a workload and set up multiple clusters, depending on the requirements as shown in Figure 2.2.

- **Single cluster per deployment unit:** In this approach several small clusters are deployed for each deployment unit of an application. A deployment unit is a single development, test, or production instance of an application. This approach allows complete isolation of the workloads from each other.

- **Single cluster per application:** In this approach, all the instances of an application, i.e., development, testing, and production are deployed on a single cluster and each application is deployed in a separate cluster. This approach enables the customization of each cluster according to the application.

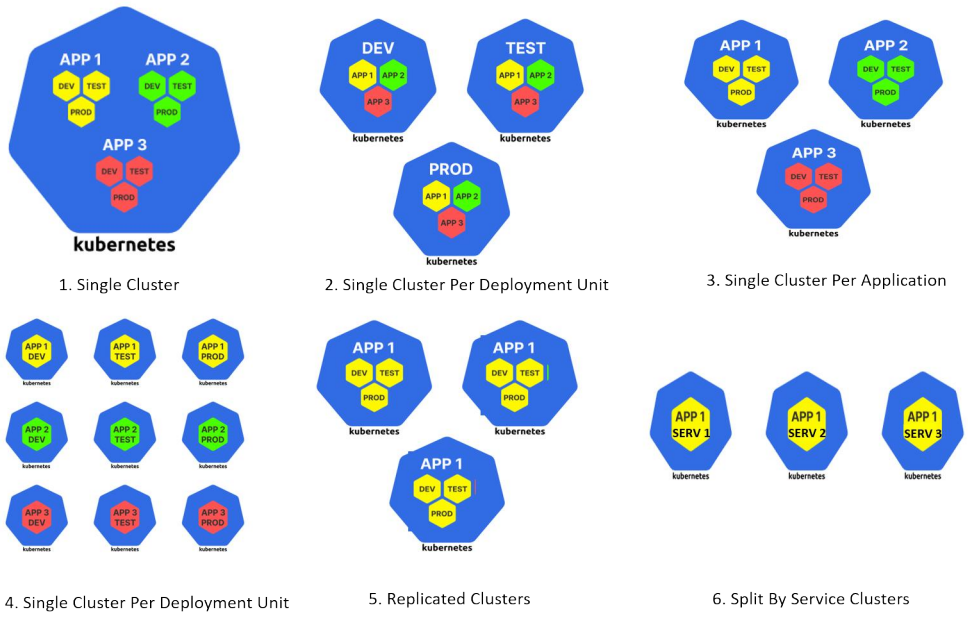

Figure 2.2: Multi-cluster Kubernetes Approaches

- **Single cluster per computing environment:** This approach isolates the environments in which the applications are developed, tested, and deployed. It allows the configuration of each environment separately and limits access to the production environment, thereby limiting the impact of an incident on the production workloads.
- **Replicated clusters:** In this approach, an application runs completely on a single cluster and its replicas are run on the other clusters. With this approach, the application can be scaled globally by having different replicas of the application available in different geographical locations. The traffic to the application can be routed to the nearest availability zone of the application to improve its response time.
- **Split by service clusters:** In this approach, an application is divided into several parts based on the services it provides, and each service is deployed in a separate cluster. This approach may be used for making compliance with regulatory requirements easier. For instance, isolating Payment Card Industry Data Security Standard (PCIDSS) compliant services from other application services. This approach enables stronger isolation between different parts of an application and allows the developers to work on individual services without disrupting the entire

application.

2.2.2 Multi-Cluster Kubernetes Configuration

There are multiple ways to configure multi-cluster Kubernetes, two of which are discussed below [46].

- **Kubernetes-Centric:** Kubernetes-centric approaches focus on modifying the existing Kubernetes configuration to have a centralized control plane that manages multiple clusters. Some examples of projects working on Kubernetes-centric approaches include KubeFed [17], Admiralty [2] and Shipper [44]. These projects will be discussed in detail in Chapter 3.
- **Network-Centric:** Network-centric approaches focus on establishing a strong network connection between multiple clusters so that they can communicate with each other. Istio service mesh [13], Network Service Mesh (NSM) [39], and Consul [7] are some examples of network-centric multi-cluster projects that are further discussed in Chapter 3.

2.2.3 Multi-Cluster Kubernetes Benefits

Multi-cluster approaches provide several benefits which are discussed below [12, 46].

- **Increased availability and performance:** The multi-cluster approach enables the users to deploy their clusters into different geographical locations for geo-redundancy. Deploying the clusters closer to the end-users improves the performance. Also, the replicas of an application on different clusters make the application highly available for end-users.
- **Isolation and multitenancy:** Multi-cluster Kubernetes provides a greater level of isolation by deploying applications and environments into separate clusters. Isolation reduces the amount of disruption caused if a specific cluster fails and makes it easier to run different operational processes on individual clusters. Also, it is relatively easier to route each tenant (end-user) to specific clusters and keep them isolated.
- **Compliance:** Multi-cluster Kubernetes reduces the scope of the cloud-specific rules and regulations that each cluster has to comply with. Therefore, it becomes easier to configure the clusters based on the rules.

- **Avoid vendor lock-in:** A multi-cluster setup makes it easier to avoid vendor lock-in and users can take advantage of the cloud capabilities and pricing offered by different vendors by deploying different clusters on different cloud platforms.
- **Improved operational readiness:** Multi-cluster reduces the operational errors and simplifies the cluster creation process and operational costs.

2.3 Kubernetes Clusters Federation

Kubernetes Clusters Federation is a mechanism through which the configuration of applications on multiple Kubernetes clusters is coordinated [52]. It schedules the workload on multiple clusters located in different regions from a single interface based on given requirements, providing low latency and high bandwidth to users [54]. Federation helps the developers in improving the responsiveness and reliability of the applications deployed on multiple clusters [58]. In a federation, the clusters are configured in a way that if one of the clusters in the federation fails, another cluster that runs the replicas of the applications on the failed cluster immediately takes over its place, thereby making the applications resilient. For instance, if an organization has an application deployed on two clusters, one is running on Google Cloud Platform in Asia and another is running on a bare-metal server in Europe, if the cluster in Asia fails, the application would still be available on the other cluster in Europe. Section 2.3.1 provides an overview of a reference Kubernetes Federation Architecture.

2.3.1 Kubernetes Federation Architecture

A sample architecture of the Kubernetes Federation is presented in Figure 2.3 . In a Federation, a single management point called manager combines several clusters. The manager coordinates the configuration of the applications on other clusters. The configuration of the application to be deployed is applied to the manager which schedules the deployment and manages the spanning clusters. This manager is a single point of failure. If it becomes unavailable, it would be impossible to deploy applications on multiple clusters from a single point and monitor the state of those clusters.

For instance, an organization has three clusters, i.e., Cluster1, Cluster2, and Cluster3. Cluster1 is running on Google Cloud Platform in Asia, Cluster2 is running on Amazon Cloud in Europe and Cluster3 is running on a

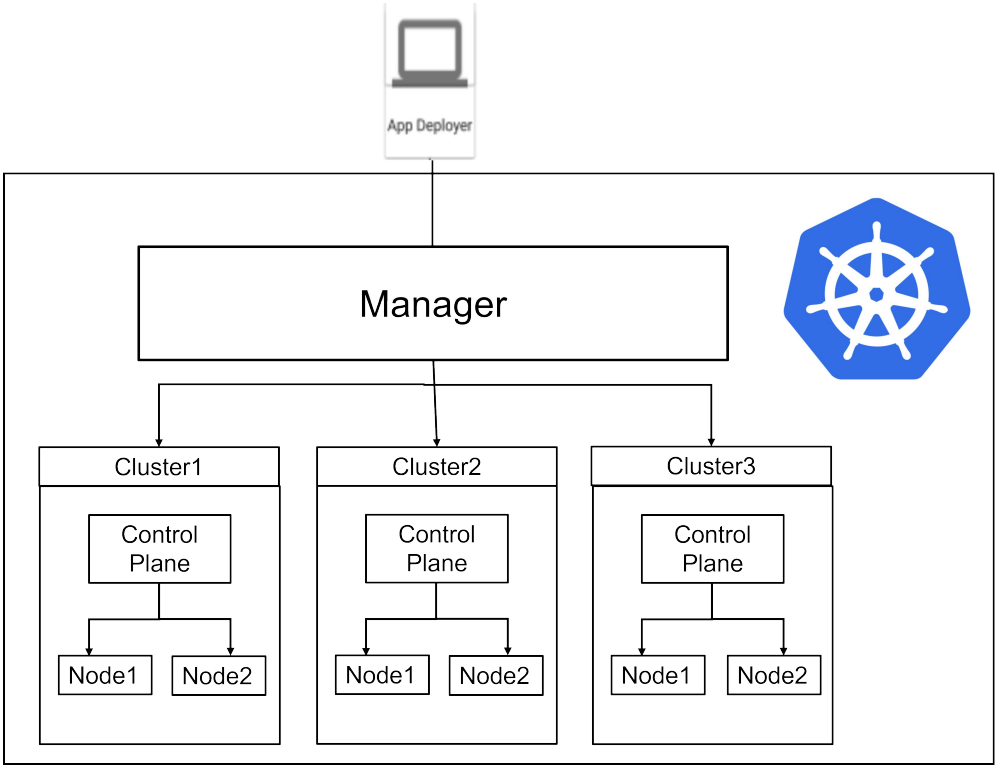

Figure 2.3: Sample Architecture for Kubernetes Federation

bare-metal server in Europe. The clusters are set up to work as a federation. The application deployer can federate the applications on Cluster2 and Cluster3 by applying the configuration of the applications on Cluster1. If Cluster1 fails, the application would still be available on Cluster2 and Cluster3. However, the application deployer would not be able to federate more applications. Also, in the absence of Cluster1, it would be difficult to monitor the desired state of the applications on Cluster2 and Cluster3.

2.4 Summary

This chapter focused on the basic concepts this thesis is based upon: Kubernetes, Multi-Cluster Kubernetes, and Kubernetes Federation. Kubernetes is the leading container orchestration system that automates the deployment, management, and scaling of applications. When Kubernetes is deployed, it provides a cluster with a control plane and several worker nodes. When we deploy applications on a single Kubernetes cluster, it has certain limitations.

For this reason, organizations have been moving towards deploying applications on multiple Kubernetes clusters. These multi-clusters can be configured in a Kubernetes-centric or a network-centric way. Kubernetes Clusters Federation provides a mechanism to manage applications on multiple Kubernetes clusters from a single management point.

Chapter 3

Kubernetes Federation Projects

The previous chapter overviewed multi-cluster Kubernetes and two ways multiple Kubernetes clusters can be configured i.e., Kubernetes-centric and Network-centric. Additionally, the previous chapter described Kubernetes Clusters Federation and its sample architecture. This chapter overviews different projects offering Kubernetes Clusters Federation solutions along with their advantages and limitations available. Afterwards, this chapter describes the main architecture of KubeFed, the second iteration of federation efforts from Kubernetes upon which this thesis is based.

The projects working on Kubernetes Clusters Federation has been divided into two categories, i.e., Network-centric and Kubernetes-centric.

3.1 Network-Centric

3.1.1 ClusterMesh Cilium

ClusterMesh [6] is a multi-cluster implementation of Cilium, a network plugin providing network connectivity along with load balancing between the application deployed on containers. ClusterMesh extends cilium by providing pod IP routing, transparent service discovery, network policy enforcement, and transparent encryption services across multiple clusters. Each of these services is built in a separate layer and can be implemented separately depending on the requirements of the users. Each cluster in a ClusterMesh has its own etcd server used to maintain its state and communicate with other clusters via etcd proxies. ClusterMesh can be used in several ways and provides many benefits. First, it can be used to ensure the high availability of the Kubernetes Clusters. The services deployed on the clusters can be replicated onto clusters in different availability zones. In case of failure of

one cluster, the requests can be redirected to the replica cluster. Second, it can be used to provide shared services to multiple tenants. Third, the stateless and stateful services can be split across multiple clusters making migration of the services from one cloud platform to another one easier. One of the major limitations of cluster mesh is that it requires cilium managed etcd which is used to guarantee quorum, create certificates and manage compaction. Also, cluster mesh requires cilium as CNI which is complicated to set up as compared to other CNI plugins [5].

3.1.2 Submariner

Submariner [45] is an open-source project providing layer-3 network connectivity between Kubernetes clusters in a secure and performant way. It creates network tunnels and routes to enable network communications between the applications deployed on multiple clusters without setting up a NodePort, ingress controller, or load balancer. It enables geo-redundancy, scaling, and fault tolerance by allowing the pods and services on different clusters to directly communicate with each other in both on-premise and cloud platforms. Submariner provides several features including compatibility with existing Kubernetes clusters, encrypted network paths using Internet Protocol Security (IPsec) a secure network protocol, centralized broker accessible by all the clusters, flexible service discovery, and compatibility with other Container Network Interface (CNI) plugins. There are several limitations of Submariner. First, it requires the minimum Kubernetes version 1.17. Second, it only supports kube-proxy in iptables mode and not in IP virtual server mode. Third, if the Kubernetes clusters are deployed using Calico plug-in, then additional configurations are needed to make them compatible with Submariner.

3.1.3 Network Service Mesh

There are many limitations of the Kubernetes network, such as lacking advanced L2/L3 connectivity, inability to meet the dynamic requirements of pods, and lacking multi-cloud/cross-cluster network connectivity support [30]. Network Service Mesh (NSM) [39] is a network service. A network service is a network function, such as bridge, router, firewall, and VPN gateway that provides services at L2/L3 layer and processes and forward packets. NSM provides L2/L3 connectivity such as virtual L2 networks, virtual L3 networks, VPNs, firewalls, and DPI, etc. on applications deployed in Kubernetes clusters. NSM is a standalone cloud-native solution that can be deployed in and outside of a Kubernetes cluster and supports cross-cluster

and multi-cloud networking. NSM contains several components, such as Network Service Endpoints (NSE) to accept client requests for network service, Network Service Clients (NSC) clients to initiate the request for a network service, Network Service Registry (NSR)–the registry of NSM components, and Network Service Manager (NSmgr)–the control plane of NSM, and NSM Forwarder to provide end-to-end connections. There are many benefits of NSM. First, it provides dynamic and advanced network services which are not available with Kubernetes CNI plugins. Also, it is easier to optimize and can work well with high-speed and low latency applications. Moreover, it can also work with service meshes and Software-Defined Networking for providing comprehensive network capabilities to Kubernetes clusters. However, NSM is limited to only networking capabilities and does not provide orchestration functionality.

3.1.4 Istio Multicluster

Istio [13] is a service mesh that provides an interface to control how microservices interact and share data with each other. A service mesh is an infrastructure layer that can be added to applications to monitor and manage the network of microservices under a single administration. Istio contains two components, i.e., *the control plane* and *the data plane*. In the data plane, an Envoy proxy is added as a sidecar proxy along with each microservice that intercepts the incoming and outgoing traffic from and to other proxies. An Envoy proxy is an edge and service proxy that adds observability, resiliency, and routing to the service. The control plane is responsible for managing and configuring these Envoy proxies. The clusters in Kubernetes are all connected to the Istio control plane and the Envoy communicates with the control plane to create a service mesh. Istio provides a dedicated infrastructure layer that can be transparently added to the top of your distributed applications. It has several features, such as TLS encrypted communication with native certificate management, strong identity-based authorization and authentication, log monitoring, fine-grained traffic control, and load balancing. Istio provides a nice ecosystem that facilitates the communication between the microservices and provides features for configuring, managing, and monitoring the clusters. It makes the application deployment and testing easier and it is easier to diagnose errors. The traffic between the services can be easily monitored and it offers valuable insights. There are also certain limitations of Istio that make it difficult to implement, such as outdated documentation, lack of instructions on how to do different things, and mismatch between versions of products. Istio also adds complexity to the system due to an additional layer. Additionally, a single control plane is a single point

of failure that would fail the service mesh if it goes down.

3.1.5 Consul

Consul [7] is a service mesh like Istio that enables communications between the microservices and provides services with service discovery configuration and segmentation functionality. It requires a data plane and supports both proxy and native integration models for service meshes. Although Consul has a built-in proxy, any 3rd party proxy, such as Envoy, can also be used with it. The main features of Consul include service discovery for clients, health checks for cluster health, TLS encrypted communication, and support for multiple data centers. Consul provides its own key-value data store with a simple HTTP API used for storing data. Consul provides several advantages. It does not have a central control plane so changes can be made to components at the edge. It also allows the users to keep half of the services in the virtual machines and half of the services in Kubernetes that adds versatility. Consul's architecture allows it to be installed to any system without requiring any additional system. Although consul is extremely lightweight and streamlined, it has several limitations. It enforces identity and authorization to layer 4 only. Also, despite providing TLS it does not support native certificate management like Istio and the certificates cannot be manually changed or modified if they are compromised.

3.2 Kubernetes-Centric

3.2.1 Shipper

Shipper [44] provides the application deployers a high-level API that allows them to use kubectl for managing Kubernetes objects with complex rollout strategies on multiple clusters. It has two types of clusters, i.e., *management cluster* and *application clusters*. Shipper itself runs on the management cluster that stores the cluster objects and secrets that allows it to connect to the application clusters. The application clusters are where Shipper deploys the applications for the application deployers. Shipper does not require any component to be downloaded to the application clusters and allows the application deployers to deploy their applications in different geographical locations and cloud platforms without any compliance issues. Shipper keeps a record of all the rollout steps and allows the application deployers to abort or revert any step. The users can additionally customize the rollout strategy for each application cluster and customize the speed/risk according to their

requirements. Shipper provides several benefits, such as it minimizes the risk for deployment by allowing the application deployers to revert back the deployments to an earlier state. Moreover, it can keep a rollout on a specific step for up to 10 hours which makes it easier to abort a step. Shipper also has certain limitations, such as it cannot perform rollouts for StatefulSets, HorizontalPodAutoscalers, and bare ReplicaSets. Moreover, it has tight integration with Helm and is designed to take Helm as input so the application deployer cannot use Kubernetes vanilla resources. Also, Shipper ensures that all clusters in the rollout strategy are rolled out at the same state due to which it cannot roll out applications cluster by cluster. Additionally, the management cluster where Shipper runs is a single point of failure. If the management cluster fails, the application deployer will not be able to roll out applications on application clusters.

3.2.2 Admiralty

Admiralty [2] provides a set of Kubernetes controllers to schedule workloads across multiple clusters. There are two types of clusters in admiralty, i.e., *source clusters* and *target clusters*. Source clusters are the ones where we configure the pods to be scheduled and target clusters are the ones where the pods are deployed. Admiralty lets the application deployer federate the resources without rewriting the deployments. Whenever a pod is scheduled in a source cluster, admiralty creates a dummy pod that sleeps. Admiralty uses a mutating pod admission webhook to intercept the pod creation process and starts another round of scheduling that would make the pod placement decision for federation and deploy the pods in the target clusters. Despite being a new platform admiralty has several advantages. It provides high availability and centralized access control and audio logs. Moreover, there is no federation-specific configuration required. Also, it has active disaster recovery and allows to scale clusters easily. However, it also has the limitation of a single point of failure. If a source cluster fails, the applications cannot be deployed to the target clusters of those source clusters.

3.2.3 KubeFed

The Kubernetes Federation approach is introduced and managed by Multi-cluster Special Interest Group that focuses on issues related to multi-cluster management in Kubernetes. Kubernetes Federation v1 was maintained by the core Kubernetes team. It is very similar to the actual Kubernetes and lets the user take advantage of Kubernetes annotations to federate resources by modifying the Kubernetes API. Kubernetes Federation v1 has many lim-

itations, such as difficulty in modifying Kubernetes API at the cluster level, limited flexibility, and no settled path to general availability, due to which it is now deprecated [27].

KubeFed [17] or Kubernetes Federation v2 is the second iteration of the federation efforts from the Kubernetes community. It is the official project of Multicluster Special Interest Group. KubeFed allows the application deployers to deploy applications on multiple clusters from a single point. The process of deploying applications to multiple clusters from a single interface is known as Federation. In KubeFed, a single cluster known as the host cluster combines several other clusters. The host cluster acts as a manager and coordinates the configuration of the applications on other clusters. The other clusters that join the host cluster are called member clusters. The host cluster can also act as the member cluster and execute actual workloads. The host cluster exposes the Federation API, schedules the deployment, and manages the spanning clusters.

The management cluster contains two main components, i.e., *the Federation API* and *Federation Controller Manager* as shown in Figure 3.1.

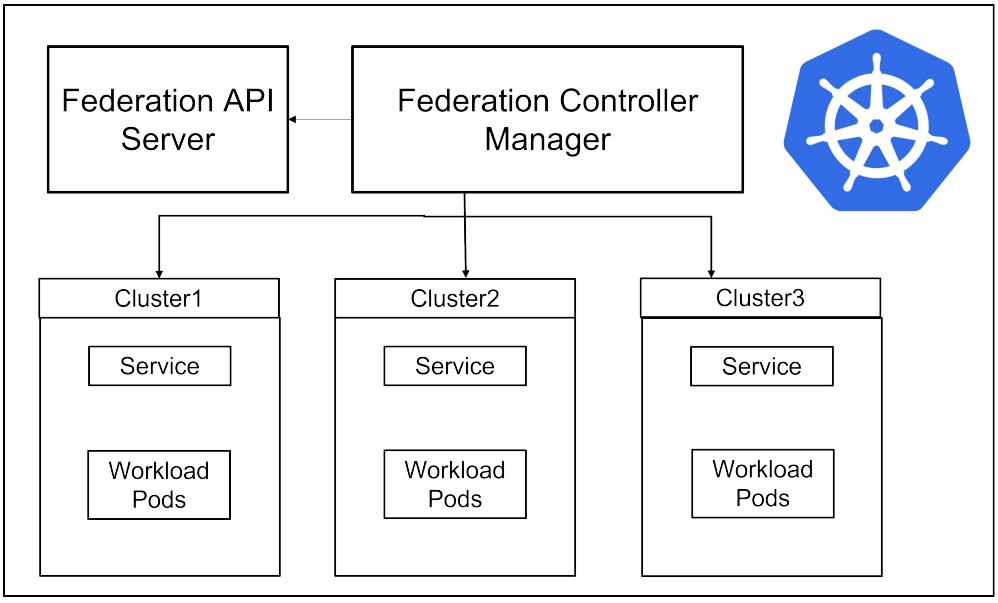

Figure 3.1: KubeFed Architecture [58]

- **Federation Controller Manager:** is similar to Kubernetes controller manager that acts as a manager and implements the non-terminating control loops. It watches the state of the clusters in the federation and makes or requests changes to maintain the desired state of the clusters.

KubeFed uses a push reconciler method to synchronize the state of the federated resources in the host cluster with the Kubernetes resources in the member clusters.

- **Federation API Server:** The API server is exposed by the management cluster. The federation API is similar to the Kubernetes API through which the controller manager communicates to the clusters.

  KubeFed makes it easier to configure applications on multiple clusters. The application deployers configure the applications on the host cluster and specify the placements policies and the resources that would be scheduled on the member clusters. Also, it allows the application deployer to abstract the applications from the cloud provider and makes it easier to move containers from one cloud platform to another. However, the project is not yet mature and has a major limitation. In KubeFed the application deployer applies the configuration of the application to be federated onto the host cluster and the KubeFed controller pushes that onto the member cluster. If the Kubernetes cluster on which KubeFed is running fails, the KubeFed host cluster would fail and the application deployer would not be able to federate any resources. Moreover, the controller manager would not be able to ensure the desired state of the cluster when the host cluster would be unavailable.

3.3 Summary

This chapter described several projects offering Kubernetes Clusters Federation solutions along with their advantages and shortcomings. In Kubernetes Federation, a single management point manages the federation of multiple clusters. The management point is a single point of failure. If the management point fails, the application deployer would not be able to federate further applications. This limitation has been observed in a number of projects discussed in this chapter including KubeFed, Shipper, Istio, and Admiralty. Therefore, this thesis aims to address this limitation and proposes a solution that eliminates the single point of failure and provides resiliency in Kubernetes Federation management. For the design and implementation of the solution, this thesis focuses on KubeFed since it is the official project of the Kubernetes community. Moreover, KubeFed is more widely adopted, has better support from the community, and has active contributors as compared to the other projects.

Chapter 4

Resiliency and Consensus

The previous chapter described several projects providing Kubernetes Clusters Federation solutions including their benefits and limitations. Additionally, it explained the main architecture of KubeFed, the second iteration of the federation efforts from the Kubernetes community along with its limitation of having a single point of failure. This thesis addresses the issue of the single point of failure by designing a solution to achieve resiliency in KubeFed. This chapter defines resiliency and explains how it can be achieved using two different strategies. In addition, it explains redundancy and consensus protocols which are the building blocks of this thesis design.

4.1 Resiliency

Resiliency refers to the use of strategies to ensure the high availability of a system. A system is said to be resilient if withstands a number of component failures and continues operating after a disruption, such as power failure, and network disconnectivity. Resiliency is often achieved with the help of redundant components to eliminate the single point of failure [9]. If one component fails or is unavailable due to any disruption the redundant component operates in place of the failed component.

Resiliency is often measured in terms of the availability of a system at a given point, such as calculating the delays or frequency of faults, and the time to recover from faults [42]. The two important factors used to measure resiliency are *Mean Time To Failure (MTTF)*, i.e., the average time taken by a system to fail, and *Mean Time To Recovery(MTTR)*, i.e., the average time taken by a system to recover. An optimal configuration of a resilient system ensures high MTTF and low MTTR.

There are many strategies that can be employed to achieve high avail-

ability for ensuring the resiliency of a system, two of which are as follows [1].

- **Active-Active Components:** One of the approaches to make a system resilient is to deploy the components as active-active. In this approach instead of using a single component, at least two components both providing the same service are used for serving the client. Both components are set up for redundancy and the individual configurations/services are redundant on both components. When one of those components fails, the other continues to serve the clients of both components. This approach is used in load balancing scenarios where the load is spread among a number of components.
- **Active-Passive Components:** In the active-passive approach, similar to the active-active approach there is more than one component. However, as the name implies not all the components are in the active mode. In this approach, one of the components is in the active mode and serves the clients while the other components remain in the passive mode until the main component goes down or is disconnected. Like in the active-active approach it is important for both components in the active-passive approach to have redundancy and similar configuration so that the client is unable to identify the difference when the passive component takes over the place of the active one.

4.2 Redundancy and Consensus Protocols

Redundancy refers to the intentional replication of a system's components. It helps us to enhance resiliency and ensures that the system continues to operate after a failure or fault. A resilient system works as a coherent system and is able to survive the failure of a number of components. An important requirement of a redundant system is to ensure that all the components are synchronized and share the same state. This behavior is achieved using consensus protocols [66] that help us in achieving a fault-tolerant system by replicating the data on multiple components. Consensus protocols implement log abstraction which contains the trace of events in the order of execution. These logs are replicated on all the components and appear as if there is a single state machine. Consensus protocols satisfy the following two properties to achieve this.

- **Consistency:** All components should have the exact same logs.

- **Liveliness:** All the events or client requests should be immediately added to the logs.

These protocols are known as consensus protocols or algorithms because, in order to add any value to the logs, the majority of the components should reach a consensus. The values are proposed by one of the components and the other components vote on the value to reach a consensus. These algorithms guarantee safety [60] by ensuring that only a single value proposed by one of the components is chosen and a value is not committed in the logs unless a consensus of the majority of the components has been received. Consensus algorithms usually work by selecting a leader from the components. All the values to be committed are sent to the leader by the clients that propose the value to the other components. When the leader receives the consensus of the majority of the components, the value is committed to the logs. If the leader component fails, another node is chosen as the leader [53] The leader is usually elected using the votes of the majority of the components. The leader election mechanism ensures that the new leader will not overwrite any of the previously committed logs. There are many consensus algorithms. In this thesis, three consensus algorithms have been discussed to evaluate their feasibility for the thesis design.

4.2.1 Paxos

Paxos [59] is the first rigorously proved consensus algorithm used to achieve consensus among a number of components, also referred to as nodes that communicate via the asynchronous system. Paxos runs a two-phased commit approach. In the two-phase commit approach, the database commit operation is split into two phases. In the first phase, a coordinating node sends a request to all the nodes to write the data to its logs. Once the coordinating node receives the positive response from the majority of the nodes, it sends the commit instructions to the nodes to commit the data. After committing the data each node again sends a confirmation to the coordinator. In Paxos, the nodes take any or all of the following roles as shown in Figure 4.1.

- **Proposer:** This node receives the value from the client and proposes the value to other nodes. There can be multiple proposers in Paxos.
- **Acceptor:** This node accepts the value proposed by the proposers. Also, it informs the proposers if a value proposed by some other proposers was accepted.
- **Learners:** This node announces the result.

Paxos is a majority-win proposal that requires the votes of the majority of the nodes to reach a consensus. In Paxos, the client sends a request to write a value to the node working as a proposer. In phase 1 one of the proposers receives a value from the client, creates a proposal id, and sends the value to acceptors asking if they have already accepted a value. The acceptors would check if they already have a proposal id in their logs which is greater than the proposal id sent by the current proposer. If they already have a larger proposal id in their logs it would mean that they have already accepted a proposal from some other proposer. In case of the absence of a larger proposal id in their logs, the acceptors add the proposal id in their logs and send a promise to the proposer signifying that they are ready to accept the proposal. If the proposer does not receive a promise from the majority of the acceptors, it starts another Paxos round. If the proposer receives a promise message from more than 50% of the acceptors then in the second phase it proceeds towards getting consensus by proposing a value. If the proposal id is still larger than the one present in the logs of the acceptor and the acceptor has still not accepted any other proposal with a larger proposal id then the acceptor would accept the value and append it in the logs. The accepted value is then propagated to the proposer as well as the learners by the acceptors. The learners would then announce the results. Paxos needs at least 2n+1 nodes to survive the failure of n nodes.

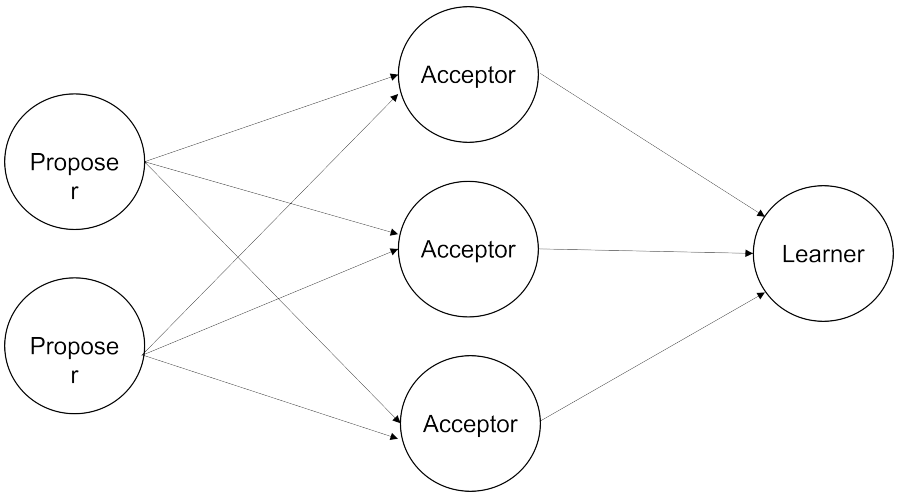

Figure 4.1: Paxos Architecture

With the current approach in Paxos, there is a possibility to reach a livelock where different proposers keep sending new values and Paxos makes no progress. This can be avoided by selecting a single proposer known as

the leader using leader election. The leader election can be performed using algorithms like the Bully algorithm. Paxos is designed to agree on a single value only once without modifying it. In order to enable it to agree on multiple values repeatedly for state machine replication and replicated logs, Paxos can be executed multiple times known as MultiPaxos [68]. Although Paxos is the first proven algorithm, it is difficult to understand. Paxos is mostly theoretical and does not consider many real-life engineering issues. Additionally, it is hard to implement and safely optimize. Moreover, only the simple Paxos is proven to be correct. The subsequent algorithms, such as MultiPaxos, are still unproven.

4.2.2 Zookeeper Atomic Broadcast

Zookeeper Atomic Broadcast (ZAB) [57] is a consensus protocol that provides strong consistency for state machine replication. This protocol is developed by the Zookeeper developers and is used in Zookeeper. Zookeeper is an open-source server by Apache that is used for storing information, such as configuration, naming conventions, and synchronization, for distributed clusters. In Zookeeper the nodes can be in one of the following three states as shown in Figure 4.2.

- **Leader:** The state in which a node receives the write requests from the clients and coordinates data replication.
- **Follower:** The node that receives data replication requests from the client.
- **Election:** The node that is participating in the leader election.

The leader election is performed in Zookeeper by assigning an ephemeral sequential number to the nodes and the node having the lowest sequential number is elected as the leader.

Every leader in Zookeeper contains an epoch number that is the unique number of the leader. In addition, whenever the leader receives a client request it generates a sequence number. ZAB constitutes a new number `zxid` consisting of an epoch number and a sequence number which is used to order client requests and is stored with each log entry. The clients write data on any of the nodes and the write request is then forwarded to the leader. ZAB also uses a variation of the two-phased commit approach to replicate data. In the first phase, the leader prepares a proposal with `zxid`, writes data to its log, and sends a write request to its followers. When the leader receives the confirmation from the majority of the followers then in the

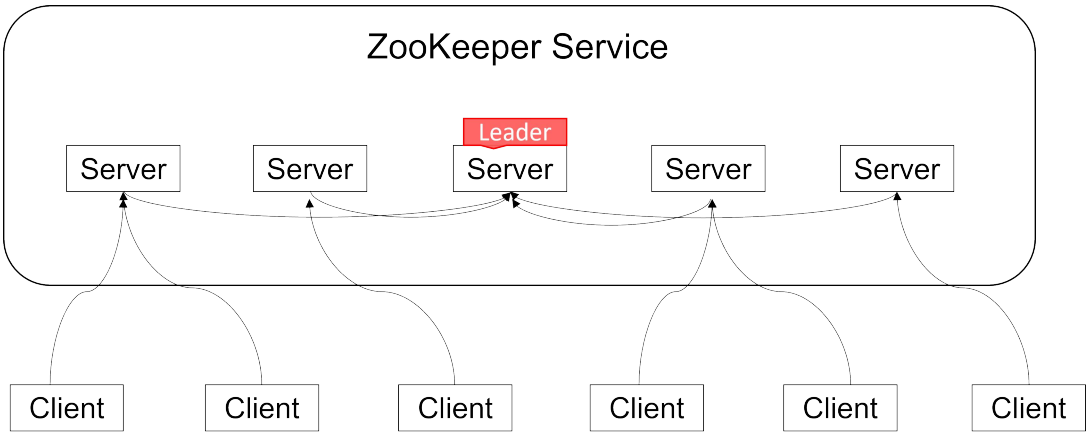

Figure 4.2: Zookeeper Architecture

second phase the leader sends commit instructions to the followers and all the nodes including the leader append the data in the logs. Although ZAB is strongly consistent, it is not a widely used protocol due to its limitations of being dependent on Zookeeper.

4.2.3 Raft

Raft [64] is another leader-based consensus algorithms used in distributed systems. Raft is often preferred over Paxos because it is comparatively easier to understand. There are three properties that make Raft different from Paxos. First, it explains leader-based consensus in a pragmatic way in the context of state machine replication which is easier to understand. Second, Raft prioritizes simplicity over performance. Third, it guarantees safety and ensures that at most one leader is elected in a given round by describing a novel approach for leader election. Also, it guarantees that an elected leader has committed logs from all the previous rounds . In Raft a node can be in one of the following three states as shown in Figure 4.3.

- **Leader:** The node that receives the client requests, gets the consensus of the other nodes and adds entries to the logs.
- **Candidate:** The state in which a node is contenting to become a leader
- **Follower:** The state in which the node receives requests for votes to elect the leader and the proposals for committing the value.

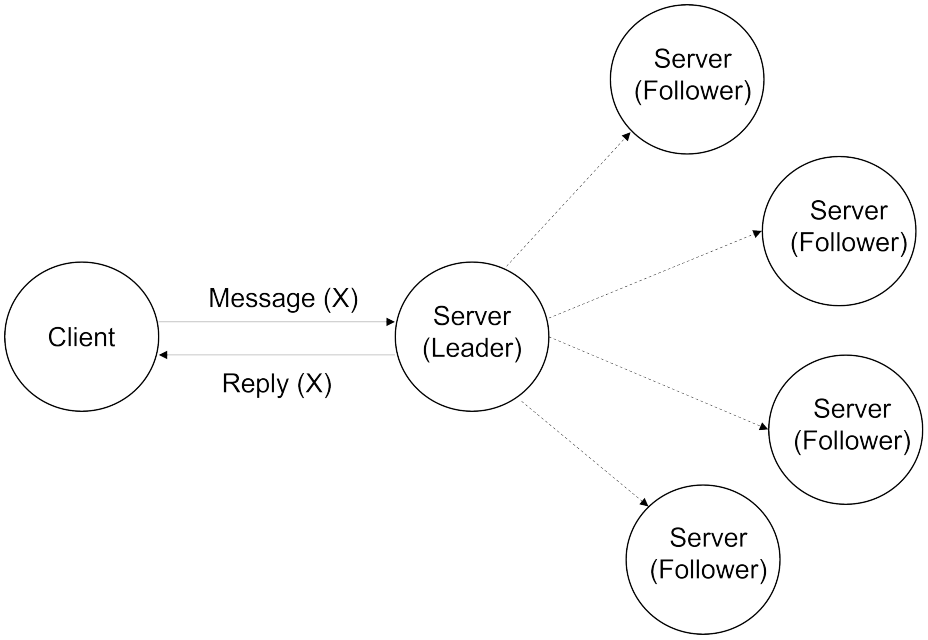

Figure 4.3: Raft Architecture

Raft algorithm simplifies the leader election process. Initially, all the nodes are initiated in the follower state. Each node has a random timeout value between 150 to 300 milliseconds. A node that first times out and observes that there is no active leader transitions to the candidate state and requests votes from other nodes to be elected as the leader. If the candidate gets the majority of the votes it is selected as the leader. The leader sends log entries to the followers. In Raft, the proposal id is referred to as term number. Raft also uses a two-phase commit approach. In the first phase, when the leader receives a request from a client, it creates a unique term number by incrementing the term and sending requests to all the followers to add that entry in the log. The followers check their logs to verify if they have any entries in the log with a greater term number. If not, they reply to the leader in the affirmative. The leader then sends commit instructions to the followers. In Raft, initially, the term is 0 on each of the nodes. Whenever a leader sends a request to the follower to write data it increases the term and sends it. The values are added in the logs along with their term number. If a leader fails to send a heartbeat to the follower within the follower's timeout duration, the follower will increase the latest term number, transition to candidate state, and request for votes. However, the follower will not vote for a candidate if the term number of the candidate is lower than the term

number of the follower.

4.3 Summary

This chapter focused on resiliency which refers to the strategy used to ensure the high availability of a system. Resiliency can be achieved by using two strategies that replicate the components of a system. In the Active-Active approach, all the replicas serve the clients simultaneously. In the active-passive approach when the active component fails the passive component takes over the place of the active component. The replicas or the redundant components should be synchronized and share the same state. Redundancy can be achieved using consensus algorithms that guarantee safety and consistency and ensure that a value is not committed unless it is accepted by more than 50% of the components in a system. Three consensus algorithms including Paxos, ZAB, and Raft were discussed in this chapter. Paxos provides a single value consensus and is difficult to understand. Also, ZAB is dependent on Zookeeper. On the other hand, Raft is easy to understand, simplifies leader election mechanism, and prioritizes simplicity over performance. Moreover, it guarantees safety by ensuring that there is at most one leader elected in each term and an elected leader has committed logs from all the previous terms. Raft is also more widely used and has several well-maintained library implementations in different languages. Due to the limitations of Paxos and ZAB and the advantages of Raft over them, Raft has been chosen in the thesis design for implementation.

Chapter 5

Design and Implementation

The previous chapter described two approaches for achieving high availability to ensure resiliency. Moreover, it explained how a system can be made resilient by making its components redundant so that whenever one component goes down, the redundant component continues to serve the clients. Additionally, it described how consensus protocols can be used for state machine replication for having redundant components. This chapter describes how these concepts can be used in this thesis implementation to achieve resiliency in KubeFed federation management. Specifically, its first section revises the problem statement and overviews the proposed solution. The second section explains the list of requirements to be fulfilled to implement that solution. The third section explains how the Proof of Concept (POC) designed in this thesis fulfills these requirements. The fourth section describes the architecture of the POC. The fifth section explains the flow of the solution POC. Finally, the last section presents the summary of the chapter.

5.1 Proposed Solution

In Kubefed, a central management point known as the host cluster manages the application deployment on multiple clusters. These clusters join the host cluster and are known as the member clusters. The application deployer communicates with the host cluster and applies the configuration information of the applications to be deployed on the member clusters. The host cluster then propagates the configuration of the applications to the member clusters and ensures that the applications on the member clusters are in the desired state using the KubeFed controller manager.

Since the host cluster alone manages the application deployment on the member clusters, this host cluster is a single point of failure as shown in

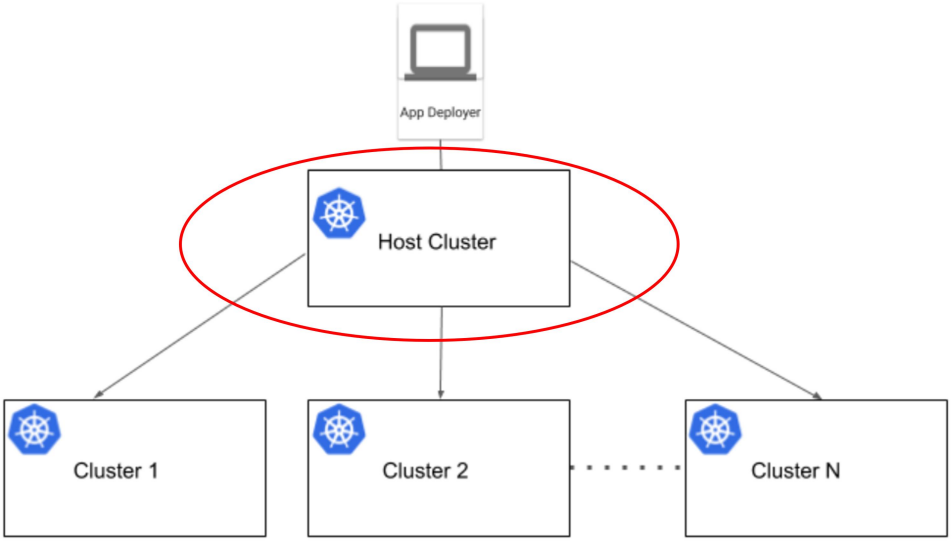

Figure 5.1: Single Point of Failure in KubeFed

Figure 5.1. If the host cluster fails or gets disconnected from the network, the workloads on the member clusters would not be disrupted and the application deployer can still interact with the individual member clusters using the Kubernetes API. However, the application deployer would not be able to federate more applications on the member clusters from a single management point. Additionally in the absence of the KubeFed controller manager, KubeFed would not be able to monitor the member clusters and ensure their desired state.

Lars Lasrron et al. [61] tried to address this limitation using a shared database of conflict-free replicated data types. In their research, several active KubeFed host clusters having access to the shared database are used. All the KubeFed host clusters can modify the state of the KubeFed objects. These clusters communicate via the shared database and are always aware of the state of the objects due to the common database. Their paper lacks the technical details of the methodology used. Moreover, the evaluation of the feasibility of their proposed solution is not published making it difficult to base any solution on the top of their research.

Therefore, this thesis takes a different approach and aims to achieve resiliency in Kubernetes Federation management by deploying active-passive redundant KubeFed host clusters. Instead of deploying a single host cluster, three or more host clusters are deployed. One cluster is selected as the leader cluster and works in the active mode whereas the remaining clusters work as

the follower clusters as shown in Figure 5.2.

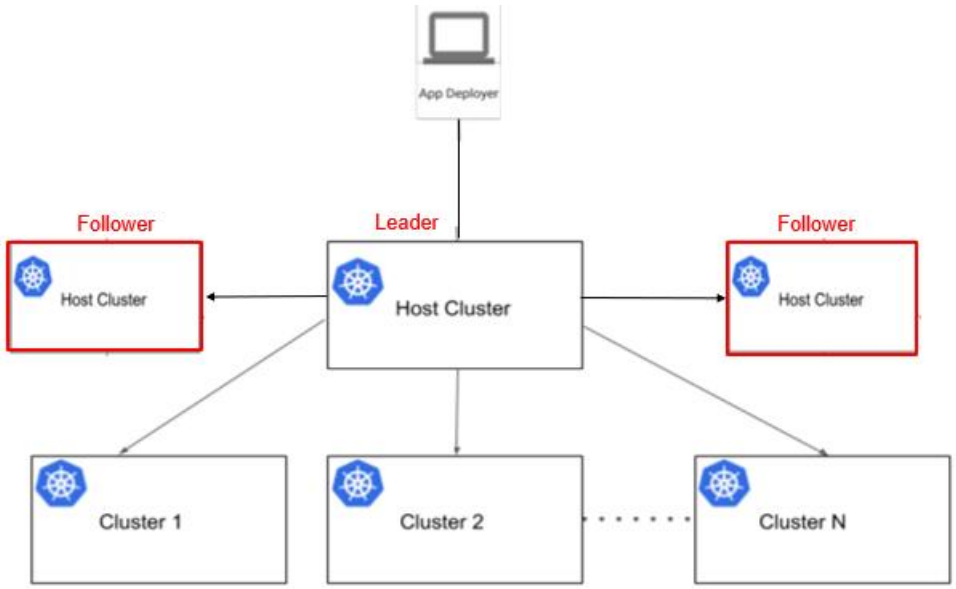

Figure 5.2: Redundant KubeFed Host Clusters

The leader cluster serves the application deployers and replicates the configuration information of the federated applications onto the followers. If the active leader fails or gets disconnected from the network, one of the follower clusters would be elected as the leader and start working as the new KubeFed host cluster. The application deployer would then be able to communicate with the new KubeFed host cluster to deploy the resources as shown in Figure 5.3. This solution would make KubeFed resilient by ensuring the high availability of the KubeFed host cluster.

There are several requirements that need to be fulfilled to implement this solution. The next section describes those requirements in a sequential way.

5.2 Requirements

5.2.1 Leader Election

The solution aims to deploy redundant KubeFed host clusters. These clusters are deployed in the active-passive mode where one cluster actively takes the leadership and is responsible for serving the application deployers and replicating the data onto the follower nodes. Therefore, an important requirement is to select one host cluster as the active host cluster. To implement this,

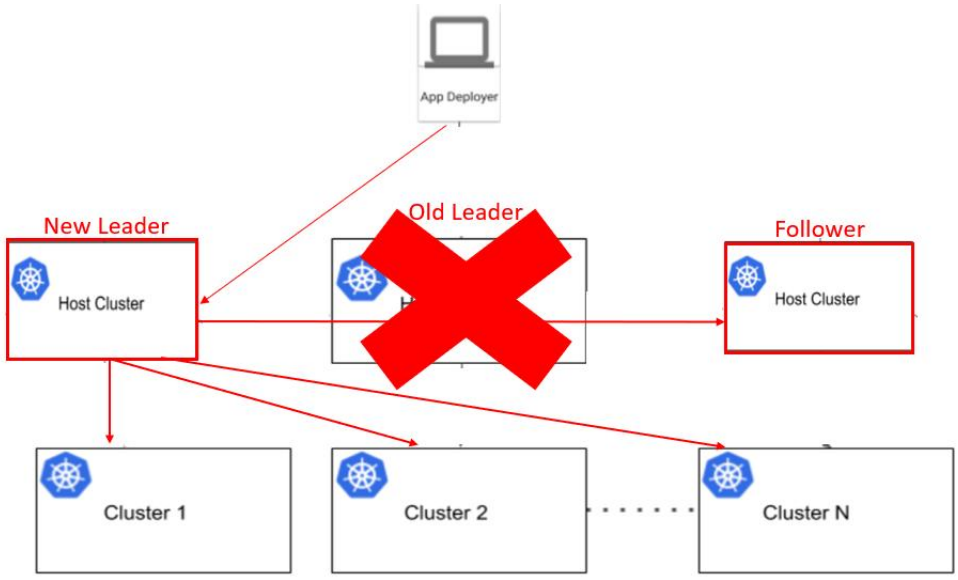

Figure 5.3: New KubeFed Host Cluster

it is required to have a leader election mechanism that selects a leader from several KubeFed host clusters. Additionally, in case of failure of the active leader, the leader election mechanism should be able to select another cluster as the new leader.

5.2.2 Watching Federated Resources

In KubeFed, the configuration information of the applications is applied on the host cluster which is then propagated to the member clusters. Almost all the Kubernetes core objects, such as namespace, deployment, and services, can be federated. In KubeFed terminology, these objects are known as federated objects. For instance, the namespace is known as federated namespace. Similarly, deployment is known as federated deployment. The federated objects should be watched continuously for obtaining their configuration information to replicate this information onto the redundant host clusters.

5.2.3 Data Replication

KubeFed makes use of Kubernetes CRDs to federate Kubernetes objects. These CRDs are wrappers for the Kubernetes core types. Each Kubernetes core type is represented by a CRD which can be modified and used to create federated resources. Also, the resources of any custom types created

using Kubernetes CRDs can be federated using KubeFed. In order to have redundant host clusters, the leader cluster should be able to replicate the configuration information of the federated resources applied by the application deployer from the leader host cluster to the follower host clusters. An example of configuration information of federated namespace that needs to be replicated is shown Figure 5.4.

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedNamespace
metadata:
  name: test-namespace
  namespace: test-namespace
spec:
  placement:
    clusters:
    - name: cluster4
    - name: cluster5
```

Figure 5.4: Federated Namespace Configuration

5.2.4 Data Management

The configuration information about the federated resources needs to be stored on all the Kubefed host clusters along with the name, type, and namespace of the federated object. Therefore, a mechanism is required to store the data on all the host clusters.

5.2.5 Configuring Leader-Follower Host Clusters in Active-Passive Mode

The solution creates multiple redundant host clusters to achieve resiliency by deploying KubeFed on all of them. In order to ensure that only the leader Kubefed host cluster is active and federates the resources and the other KubeFed host clusters are in a passive state, all the KubeFed host clusters should be configured in a way that only the leader host cluster can federate the resources. The host clusters working as a follower should be in the passive mode and should not able to federate resources.

5.2.6 Watch for Leadership Changes

Whenever the leader host cluster fails, the follower that times out first and observes this failure would transition to candidate state and would request votes from the remaining followers. During that time period, there will not be any active leader. Afterwards, when the follower would get the majority of the votes, the follower would be transitioned to the leader state. Therefore, the leadership transitions should be monitored continuously to configure the clusters accordingly.

5.2.7 Configure New Leaders

When the leader fails and a follower takes up the leadership, the follower should start from the last known state of the previous leader. Therefore, it is important to use the backup configuration information of the federated resources to configure the federated resources on the new leader as they were configured on the previous one.

5.2.8 Informing Application Deployers

The application deployer should be able to communicate with the leader host cluster to be able to federate resources onto the member clusters. Therefore, it should be ensured that the application deployer is aware of the current leader cluster. The application deployer should be able to retrieve the information of the current leader at any point and should be aware when there is no active leader or whenever the leader changes.

The next section explains how the POC designed in this thesis implementation fulfills these requirements.

5.3 Implementing Solution

5.3.1 Leader Election and Data Replication Implementation

In section 4.2, it was discussed how consensus algorithms ensure state machine replication to achieve redundancy. Three consensus algorithms were discussed: Paxos, ZAB, and Raft. Paxos is difficult to understand and provides only one event consensus. Although some subsequent Paxos algorithms, such as Multi-Paxos, provide continuous state machine replication, these algorithms are still difficult to understand and are not proven. The ZAB

algorithm could be used instead, but it suffers from limited usability due to its dependability on Zookeeper. On the other hand, Raft is easy to understand, simplifies leader election mechanism, and prioritizes simplicity over performance. Additionally, Raft is also more widely used and has several well-maintained library implementations in different languages. Due to the limitations of Paxos and ZAB and the advantages of Raft over them, in the POC implementation Raft has been used for leader election and data replication. Raft is a partition tolerant consistent protocol that ensures the system consistency as long as the majority of the nodes in a system are working. The minimum number of nodes required for Raft is 3 and the maximum number of nodes is 7. However, it is recommended to have 5 nodes in a production-ready Raft setup. A 3 node Raft cluster can tolerate the failure of 1 node, a 5 node Raft cluster can tolerate the failure of 2 nodes and a 7 node Raft cluster can tolerate the failure of 3 nodes.

The algorithm has been implemented in Golang using the HashiCorp Raft library [10]. In the Raft implementation, there are two servers. First, an HTTP server is used by the nodes participating in data replication to join the cluster and to write and read the data. Second, a Raft server for intra-node communications. Initially, one of the KubeFed host clusters is started in the bootstrap mode. The HTTP server address of the bootstrap cluster will be used by the other KubeFed host clusters to join Raft. The HTTP address of the clusters is a public address. However, the Raft address is a private one and is shared by the nodes while joining the bootstrap cluster using HTTP post request. Raft nodes communicate and share information with each other using the Raft server address via Remote Procedure Calls over TCP. Two types of RPCs are used in Raft, i.e., *RequestVote* used by the candidate to request votes for leadership, and *AppendEntries* used by the leader to request the nodes to append the data in the Raft logs and send the heartbeats.

5.3.2 Watching Federated Resources

To be able to replicate the configuration information of the federated resources on the follower host clusters, any changes in the federated objects on the leader host clusters by the application deployer should be continuously monitored. This has been achieved using KubeFed dynamic client [18]. KubeFed dynamic client provides a watch method to set up a watch on all the federated resources by providing the group, version, kind, and name of the federated object. The watch observes for any kind of change in the specified object and returns a channel. A channel in Go is a medium through which two goroutines (threads) send information to each other. The channel

can be continuously monitored for any kind of event on the specific object type. The event contains the information of the object including name and other configuration information that can be fetched for replication. The solution watches the federated objects for three types of events, i.e., creation, modification, and deletion for storing those events in the Raft logs.

5.3.3 Data Management

Raft stores logs on all the nodes belonging to the Raft cluster. These logs store the data as key-value pairs. Whenever there is a change on the watched federated resource, the configuration information of the resource is extracted from the event and stored into the Raft store as a key-value pair. The key comprises the type, namespace, and name of the federated object whereas the value consists of the configuration of the federated object.

The information can be extracted from the HTTP server of any of the Raft nodes using the following curl command.

```
curl -XGET http://172.18.0.4:30003/key/
    FederatedNamespace:test-namespace1:test-
    namespace1
```

In the above example, `172.18.0.4` is the IP address of one of the KubeFed host clusters where the POC is running, `30003` is the port number of the HTTP server and `FederatedNamespace:test-namespace1:test-namespace1` is the key name.

An example of the information of a federated namespace stored as key-value pair in the store is shown in Figure 5.5. `test-namespace1` is the federated namespace that is deployed on the host cluster within the Kubernetes namespace named `test-namespace1`.

To implement the data store, the Hashicorp Raft library provides a package called Raftboltdb that implements the log store. The store is based on Bolt [4] which is a pure GO-based key-value store. Bolt is used by applications that do not need to have a fully-fledged database server like PostgreSql [40] and MySql [38]. The store can be used to store the data in-memory as well as on-disk.

```
{"FederatedNamespace:test-namespace1:test-namespace1
    ":"{'apiVersion':'types.kubefed.io/v1beta1','
    kind':'FederatedNamespace','metadata':{'
    annotations':{},'name':'test-namespace1','
    namespace':'test-namespace1'},'spec':{'placement
    ':{'clusters':[{'name':'cluster4'},{'name':'
    cluster5'}]}}}\n"}
```

Figure 5.5: Federated Namespace Key-Value Pair

5.3.4 Configuring Leader-Follower Host Clusters in Active-Passive Mode

The deployment of KubeFed on multiple clusters would deploy KubeFed controller manager on all those KubeFed host clusters. By default, the controller managers on the leader as well as the follower KubeFed host clusters would try to maintain the desired state of the member clusters according to their configuration information. This situation might lead to a conflict between the KubeFed controller managers and can cause issues with the state of the member clusters. To avoid this, it should be ensured that only the KubeFed controller manager on the leader host cluster is in the functioning state and attempts to maintain the desired state of the cluster. To achieve this, the KubeFed controller manager leader election mechanism has been used.

During the life cycle of the KubeFed controller manager, there is a possibility to have more than one instance of the controller manager running on a single KubeFed host cluster to ensure the high availability of the KubeFed controller manager. The presence of multiple instances of KubeFed controller manager on a single host cluster might lead to a conflict between multiple controller manager instances. To avoid the conflict, KubeFed uses a leader election mechanism similar to Kubernetes to make sure that only one instance of the controller manager on a cluster is functioning at a time. An instance takes the leadership role by generating a Service Endpoint and creating an annotation `"control-plane.alpha.kubernetes.io/leader"` that contains the information about the current leader under `holderIdentity`. The annotation is exposed to the other instances by saving it in the Configmap named `"kubefed-controller-manager"`. The leader instance has a `renewTime` parameter that tells the time when the leader was last renewed. Additionally, there is a `leaseDuration` parameter that tells the duration for which the leadership of the current leader is valid. The controller manager leader instance should update its `renewTime` parameter before its lease du-

ration expires to retain its leadership. The other instances will continuously watch if a Service Endpoint exists and then check if the leader has updated its `renewTime`. If any of the above conditions fails, the follower instances would start a new leader election process for the KubeFed controller manager.

The above KubeFed controller manager leader election process works on all the KubeFed host clusters individually and multiple controller manager instances run on all of them. To push the controller manager to a passive state on the follower KubeFed host clusters, this POC modifies the value of the annotation on each follower host cluster. A dummy annotation value in the Kubefed-controller-manager ConfigMap in the follower host clusters would give the controller-manager instances running on those clusters the impression that another instance of KubeFed controller present on the same host cluster is serving as the leader. This situation would thereby prevent the controller manager instances on a follower host cluster from being active.

Figure 5.6 shows an example of the value of the ConfigMap of a KubeFed host when there is an active leader. The ConfigMap has an annotation specifying that the leader instance with identity `"kubefed-controller-manager-76f66bc57f-qdjqw-7e4704dc-f276-4c1b-8cce-6e14a1833862"` is the current leader.

```
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader: '{"
        holderIdentity":"kubefed-controller-manager
        -76f66bc57f-qdjqw_7e4704dc-f276-4c1b-8cce-6
        e14a1833862","leaseDurationSeconds":15,"
        acquireTime":"2021-08-08T13:27:59Z","
        renewTime":"2021-08-08T13:36:49Z","
        leaderTransitions":0}'
  creationTimestamp: "2021-08-08T13:27:59Z"
  name: kubefed-controller-manager
  namespace: kube-federation-system
```

Figure 5.6: KubeFed Controller Manager Configmap With An Active Leader

Another example of the value of the ConfigMap of a KubeFed host is shown in Figure 5.7 when there is no active leader. The annotation value has been modified to `Nothing` for pushing the controller manager instances

into passive mode. The same behavior can also be achieved by changing the `holderIdentity` in the annotation value to a dummy value and modifying the `renewTime` to a later date. In either case, the controller manager instances on the follower host clusters would be tricked into going in a passive state even if the POC process that modifies the Configmap dies on the cluster.

```
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader:
        nothing
  creationTimestamp: "2021-06-15T10:22:24Z"
  name: kubefed-controller-manager
  namespace: kube-federation-system
```

Figure 5.7: KubeFed Controller Manager Configmap With No Leader

5.3.5 Watch for Leadership Changes

To be able to configure the leader and followers accordingly, it is required to continuously monitor the leadership changes. The solution uses a combination of pull-based and push-based approaches to watch for leadership changes. First, it sets up an observer for observing any changes in leadership and gets a channel that can be continuously watched for leadership transitions. Second, it sets up a Go Cronjob that checks the current leader value every second to verify if the current leader value has been observed and updated. If not, the Cronjob updates the value accordingly.

5.3.6 Configure New Leaders

To configure the newly elected leader host cluster to the last known state of the failed leader, the federated resources are fetched from the Bolt key-value store. To create those resources KubeFed dynamic client has been used. KubeFed Dynamic client provides methods for creating, modifying, and deleting KubeFed resources. When a candidate host cluster becomes the leader, it fetches all the objects from the Bolt store sequentially and creates, modifies, and deletes the objects accordingly.

5.3.7 Informing Application Deployers

To keep the application deployer up to date with the current leader, a ConfigMap has been used. Whenever a leader cluster would go down or a new leader would be elected, the ConfigMap named `kubefed-leader` would be updated with the current leader cluster name and IP address under `leaderData`. Two examples of kubefed-leader ConfigMap have been shown below. In Figure 5.8, cluster1 is the leader so the leader data has been updated to cluster name and its IP address. In Figure 5.9, there is no leader so leader data has been updated to None. The user can query the ConfigMap on any of the clusters, i.e., the leader as well as the follower clusters to fetch the details of the current cluster.

```
apiVersion: v1
data:
  leaderData: |-
    name: cluster1
    address: 172.18.0.2
kind: ConfigMap
metadata:
  name: kubefed-leader
  namespace: kube-federation-system
```

Figure 5.8: KubeFed Leader Configmap With An Active Leader

```
apiVersion: v1
data:
  leaderData: |-
    name: 0.0.0.0
    address: None
kind: ConfigMap
metadata:
  name: kubefed-leader
  namespace: kube-federation-system
```

Figure 5.9: KubeFed Leader Configmap With No Leader

The next section describes the overall final architecture of the proposed solution.

5.4 Architecture

Figure 5.10 shows the generic architecture of the solution with one leader and one follower host cluster. The POC would be deployed onto the clusters using Kubernetes Deployment. The app deployer interacts with the Kubernetes Federation API server for creating or modifying the federated resources. The implementation of the POC contains an HTTP server that is used by the other clusters to join the Raft cluster. The POC monitors the interaction of the application deployer and replicates it to the follower cluster using the Raft server. This replicated information is stored inside the store as key-value pairs. In addition to this, the POC ensures that the KubeFed controller manager on the follower cluster is in a passive mode. Also, each host cluster has a ConfigMap that stores the current leader information for the application deployers.

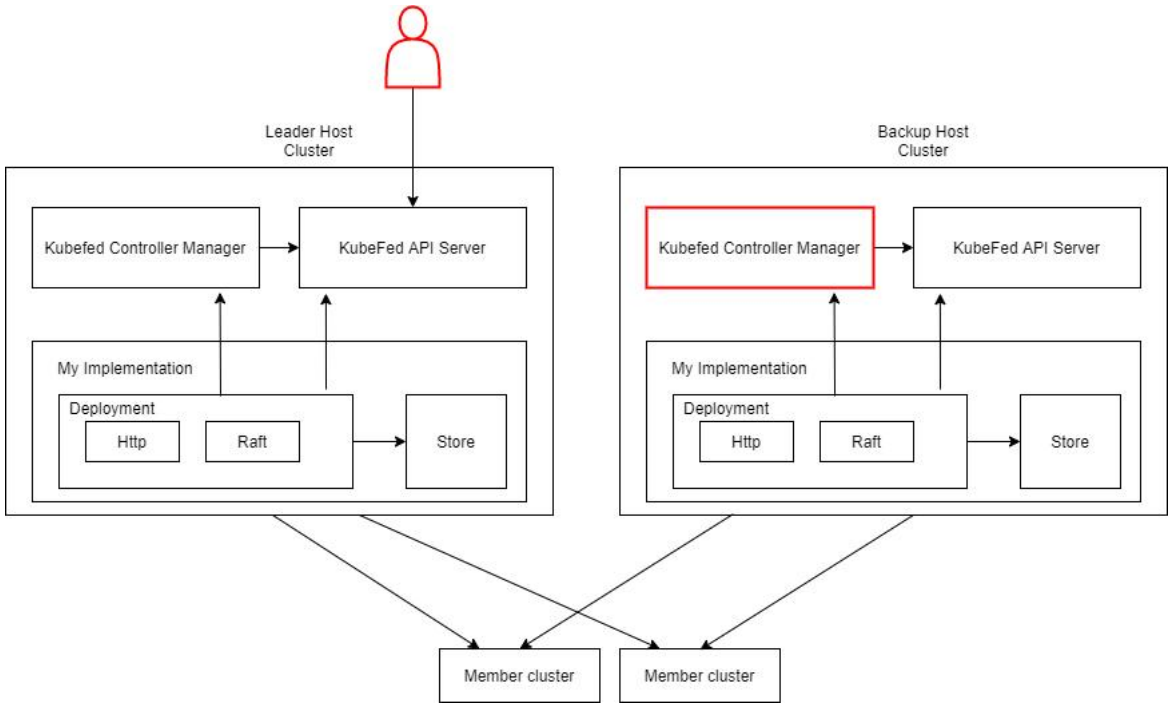

Figure 5.10: Architecture of POC

The next section describes the overall flow of the leader as well as follower clusters.

5.5 POC Flowchart

Figure 5.11 shows the flowchart of the POC. Initially, one host cluster node is started in the bootstrap mode and the other host cluster nodes join Raft using the HTTP address of the bootstrap node. Initially, all the nodes are started in the follower state and the KubeFed controller manager has stopped on all the nodes. The first node to observe the absence of leadership transitions to the candidate state and starts leader election. After the leader election, the clusters continuously monitor for leadership transition. If a cluster node becomes the leader, it starts the KubeFed controller manager and checks the logs database for any federated resource entries. If there would be any available resources, the node would create, modify or delete those resources. The node would continuously monitor the changes in the federated resources and replicate them to the follower nodes. If the node would be a follower node, it would stop the KubeFed leader and append data received from the leader to its logs. At any point, if a follower node does not get a timely heartbeat from the leader, it would transition to the candidate state for re-election.

5.6 Summary

This chapter describes how active-passive redundant KubeFed host clusters are used in this thesis to achieve resiliency in Kubernetes Federation management. There are a number of requirements that need to be fulfilled for implementing the solution. In this thesis, leader election and data replication have been implemented using the Raft consensus algorithm. For watching and modifying federated resources KubeFed dynamic client has been used. The leadership transitions have been observed with a combination of pull and push-based approaches using Raft observer and Go Cronjob. For configuring leader and follower host clusters in active-passive mode Kubefed Controller Manager leader election mechanism has been used. For implementing the log store, a bolt key-value store has been used. For informing the application deployers about the leadership transitions, one Configmap per cluster has been used.

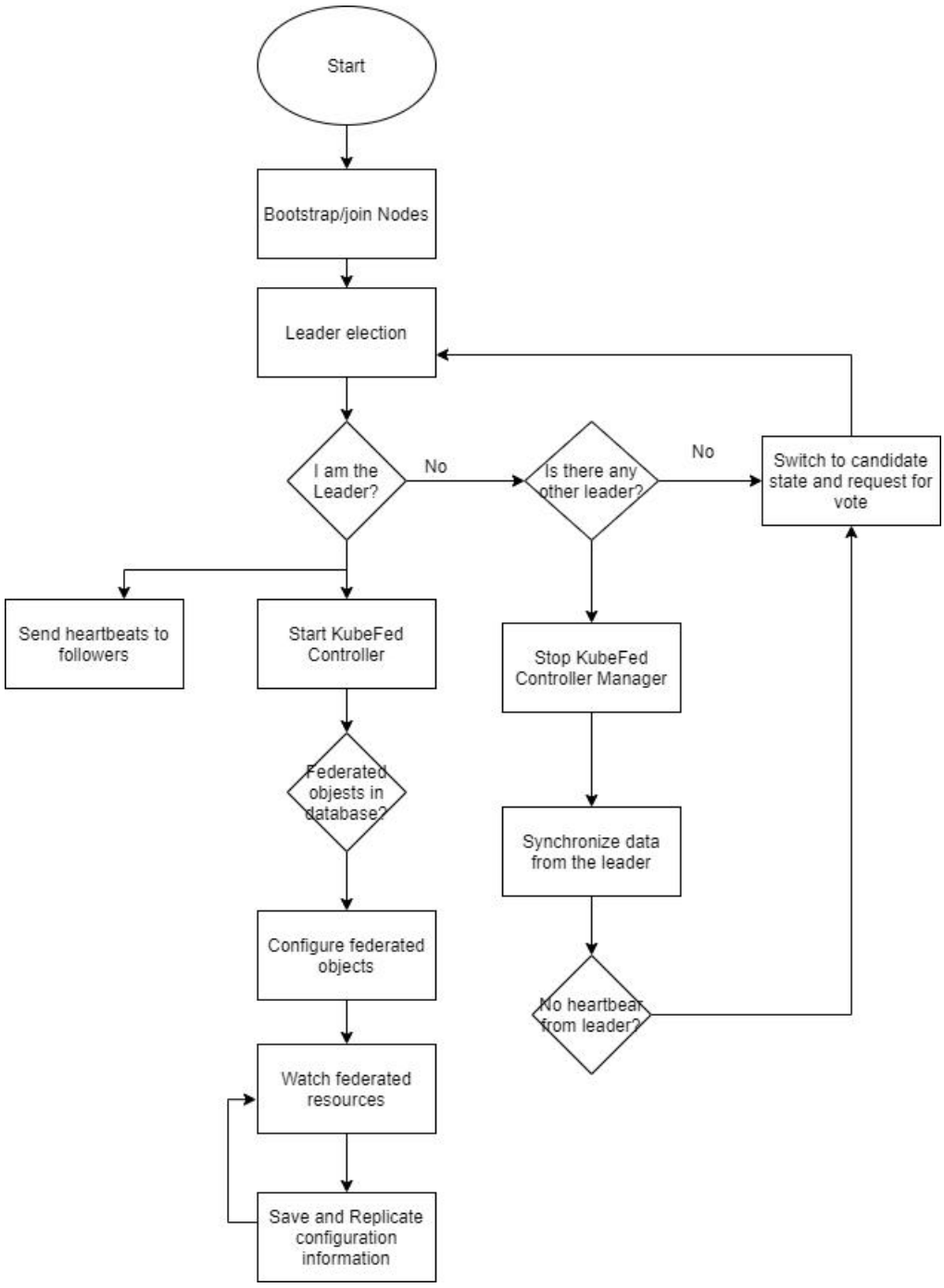

Figure 5.11: Flow Chart of POC

Chapter 6

Evaluation

The previous chapter described the proposed solution and different parts of the POC implementation based on the proposed solution. This chapter evaluates the POC presented in the previous chapter. The evaluation has been divided into two parts. As the solution aims to provide resiliency in Kubernetes federation management, in the first part it is tested for reliability and high availability. In the second part, the security of the POC is evaluated.

6.1 Reliability and High Availability

The reliability and high availability tests have been divided into two parts depending on the number of KubeFed host clusters used. The next section describes the test environment for both parts.

6.1.1 Test Environment

The POC has been implemented on a Linux virtual machine running on top of VirtualBox hosted on a Linux Computer. VirtualBox [47] is a free and open-source hypervisor software providing virtualization for x86 and x86_x64 hardware. It is a powerful tool that allows the users to run multiple guest operating systems on top of a single host operating system. The specifications of the virtual machine used in this thesis are described in Table 6.1.1.

The clusters are run on Docker containers using Kubernetes in Docker (Kind) [14]. Kind is a tool for running Kubernetes clusters on Docker containers as "nodes". Kind creates containers with a pre-built Kubernetes image. After deploying the required number of clusters, KubeFed is installed on the clusters that would serve as the leader host and the follower host clusters for the POC. Afterwards, the POC is installed on the KubeFed host clusters as a

Table 6.1: Specifications of The Virtual Machine Used

CPU	Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz
Memory	24GB
Hard disk	295GB
Operating System	Ubuntu 16.04.7 LTS
Kernal	Linux 4.15.0-142-generic x86-64

Kubernetes deployment. In addition to it, ClusterRole and ClusterBindings are created for Role-Based Access Control (RBAC) [41]. RBACs are used for allowing the POC to be able to read and modify Kubernetes and KubeFed resources. The details of the cluster deployment, KubeFed installation, POC installation, and RBACs are present in appendix A.

The next section describes the experiments and results of the first part of the reliability and high availability tests with three host clusters.

6.1.2 Three Host Clusters

For these tests, five Kubernetes clusters were deployed, i.e., cluster1, cluster2, cluster3, cluster4, and cluster5. Out of these five clusters, cluster1, cluster2, and cluster3 were configured to run KubeFed and the POC. cluster4 and cluster5 joined the KubeFed host clusters as the member clusters. Although cluster1, cluster2, and cluster3 can work as the member clusters and execute workloads, for the sake of simplicity distinct host clusters and member clusters have been used in these tests. All the required ClusterRoles and ClusterRoleBindings were created on cluster1, cluster2, and cluster3.

cluster1 was started in the bootstrap mode and cluster2, cluster3 joined the Raft node on cluster1 using the HTTP address of cluster1. After the leader election, cluster1 was elected as the leader whereas cluster2 and cluster3 were in the follower state. A series of experiments were performed which are described below.

Experiment 1: Cluster Failure

In this part, the POC was evaluated for cluster failure. A total number of four tests were performed to observe if the solution is resilient and works as expected in the case of cluster failure as shown in Table 6.2. Cluster failure was emulated by deleting the POC deployment from the cluster and pausing

Table 6.2: Experiments with Three Host Clusters

		Cluster Failure			
	Test Case	**Cluster1**	**Cluster2**	**Cluster3**	**Is resilient?**
Test 1	Leader failed	X	✓	✓	Yes
Test 2	1 follower failed	✓	X	✓	Yes
Test 3	Leader and 1 follower failed	X	X	✓	No
Test 4	both followers failed	✓	X	X	No
		Network Partition			
Test 5	Leader disconnected	✓	✓	✓	Yes
Test 6	1 Follower disconnected	✓	✓	✓	Yes
Test 7	All clusters disconnected from each other	✓	✓	✓	No

the docker container on which the deployment was running. The four tests emulated different cluster failure scenarios, i.e. leader failure, single follower failure, leader and one follower failure, and both follower failure. The results were the same as our expectations in all four cases.

In test 1, cluster1 being the leader failed, the follower node cluster2 timed out first and started the leader election process. cluster2 received the votes from itself and cluster3 and became the new leader. However, cluster2 kept trying to connect to cluster1. When cluster1 was brought back, it got the latest logs from the leader and the clusters worked as usual.

In test 2, a single follower, i.e., cluster2 failed. Since an active leader cluster1 was already there, cluster2 failure did not impact the working POC implementation. The only difference was that cluster1 tried continuously to connect back to cluster2. When cluster2 was brought back, it got the latest logs from the leader and the clusters worked as usual.

However in test 3 and test 4 two out of three Raft nodes were down which is an unexpected behavior as the Raft algorithm can only tolerate 1 node failure in a 3 node scenario, the POC stopped working and the single remaining node started leader election and waited for the other nodes to vote for the leader. When the failed clusters were brought back, a new leader election process was started and the solution elected a new leader.

From the results of the tests, it is concluded that the solution is resilient for test 1 and test 2 and not resilient for test 3 and test 4 as expected.

Experiment 2: Network Partition

In this part, the POC was evaluated for network partition. A total number of four tests were performed to observe if the solution is resilient and works

as expected in the cases of network partition as shown in Table 6.2. Network partition was emulated by using the iptables rules. An example of iptables rules is shown in Figure 6.1 that is used to block incoming and outgoing traffic on cluster1 from cluster2 (172.18.0.3) and cluster3 (172.18.0.4).

```
iptables -A INPUT -s 172.18.0.3 -j DROP
iptables -A OUTPUT -d 172.18.0.3 -j DROP

iptables -A INPUT -s 172.18.0.4 -j DROP
iptables -A OUTPUT -d 172.18.0.4 -j DROP
```

Figure 6.1: Iptables Rules for Blocking Access from cluster1 to cluster2 and cluster3

The three tests emulated different network partition scenarios, i.e. leader gets disconnected from the two followers, one follower gets disconnected from the leader and another follower, and all the three clusters get disconnected from each other. The results were the same as expected in all three cases.

In test 5, cluster1 gets disconnected from cluster2 and cluster3. Since cluster1 lost the majority of the Raft nodes, it transitioned to the candidate state and tried to get votes from cluster2 and cluster3. cluster2 and cluster3 noticed the unavailability of a leader, performed leader election, and continued working as usual. The new leader cluster3 still tried to connect to cluster1. When cluster1 was reconnected to cluster2 and cluster3, reelection started and a new leader was selected.

In test 6, cluster2 gets disconnected from cluster1 and cluster3. Since an active leader was already there, cluster2 disconnectivity did not impact the working implementation. Apart from cluster1 trying to connect to the follower cluster3 nothing was impacted. On the other hand, cluster2 transitioned to the candidate state and tried to get the majority votes to become the leader. When cluster2 was reconnected to cluster1 and cluster3, reelection started and a new leader was selected.

In test 7, since all the nodes were disconnected from each other, all nodes transitioned to the candidate state and tried to get the majority of the votes to become the leader. The POC does not work on any of the nodes if there is no active leader. When the clusters were reconnected, reelection started and the node with the latest term number was selected as the leader.

From the results of the tests, it is concluded that the solution is resilient for test 5 and test 6 and not resilient for test 7 as expected.

Experiment 3: Data Replication and Resource Federation

In this experiment, two tests were performed. In the first test, it was evaluated if the leader host cluster watches the federated resources and replicates the configuration on the follower. In the second part, it was verified that when the leader fails and the follower takes the leadership, it applies the federated resource configuration from the store and starts from the last known state of the failed leader.

In the first test, two namespaces, i.e., `test-namespace1` and `test-namespace2` were created on cluster1 and federated onto cluster4 and cluster5. The test worked as expected and the configuration of both namespaces was replicated on cluster2 and cluster3.

In the second test, cluster1 was shut down so that one of the followers could take the leadership. The test also worked as expected. cluster3 became the leader and read the logs from the bolt store and applied the configuration information of both namespaces. cluster3 continued working from the last known state of cluster1.

Experiment 4: Leader-Follower in Active Passive State

In this part, it was evaluated if the KubeFed controller manager ConfigMap in the leader and follower host cluster is updated whenever the leadership changes. The controller manager should only be active in the leader host cluster and it should be passive in the follower host clusters. This test was performed for all the scenarios listed in Experiment 1 and Experiment 2. The test worked as expected in all the scenarios. When a cluster was serving as the leader host cluster, the controller manager transitioned to active mode. However, when it was working as a follower, the controller manager ConfigMap changed the leadership annotation to push it to passive mode.

Experiment 5: Updated Leader Information

In this experiment, it was evaluated if the KubeFed leader ConfigMap is updated whenever the leadership changes. The KubeFed-leader ConfigMap should show the name and IP address of the current leader at all times. If there is no active leader, the ConfigMap should display `None`. This test was performed for all the scenarios listed in Experiment 1 and Experiment 2. The tests worked as expected in all the scenarios. When there was an active leader, the ConfigMap displayed the leader information, and when there was no active leader the ConfigMap showed `None` as the leader.

Experiment 6: Time to Recover

In this experiment, the time that a follower KubeFed cluster takes to recognize the absence of a leader and take the leadership was observed when there are three host clusters. The time is measured using the timestamps from the logs provided by the Hashicorp Raft library. The experiment was repeated for five iterations to calculate the median and the standard deviation for the recovery time of the POC with three host clusters. In the three host cluster scenario, when the leader cluster, i.e., cluster1 was taken down, the median value for the time it took for another cluster to observe that the leader is down is 2.997 seconds and the standard deviation is 0.380. Moreover, the median value for the time taken to start the leader election process and elect a new leader is 0.015 seconds and the standard deviation is 0.335. Overall, the median value for the total time for the POC to recover from a failure is 3.009 seconds and the standard deviation is 0.650 as shown in Table 6.3

The next section describes the experiments and results of the second part of the reliability and high availability tests with five host clusters.

Table 6.3: Recovery Time for The POC with Three Host Clusters

	1	2	3	4	5	Median	Standard Deviation
Leader Failure Observed (s)	3.007	3.354	2.324	2.765	2.997	2.997	0.380
New Leader Elected (s)	0.015	0.771	0.049	0.012	0.012	0.015	0.335
Total Recovery Time (s)	3.022	4.125	2.373	2.777	3.009	3.009	0.650

.

6.1.3 Five Host Clusters

In this section, six Kubernetes clusters were deployed, i.e., cluster1, cluster2, cluster3, cluster4, cluster5, and cluster6. Out of these six clusters, cluster1, cluster2, cluster3, cluster4, and cluster5 were configured to run KubeFed and POC. cluster6 joined the KubeFed host clusters as the member cluster. All the required ClusterRoles and ClusterRoleBindings were created on all the host clusters.

cluster1 was started in the bootstrap mode and cluster2, cluster3, cluster4 and cluster5 joined the Raft node on cluster1 using the HTTP address of cluster1. After the leader election, cluster1 was elected as the leader whereas cluster2, cluster3, cluster, and cluster5 were in the follower state. A series of experiments were performed to test if the solution is resilient and works as expected. The experiments are described below.

Experiment 1: Cluster Failure

In this experiment, the POC was evaluated for cluster failure similar to section 6.1.2. The solution worked as expected in all four tests. In test 1 and test 2, the solution worked identically to section 6.1.2.

However, in tests 3 and 4 since now we have 5 Raft nodes the POC is now able to handle 2 node failure. In test 3 when the leader and one follower fail, the remaining 3 nodes select another leader and the solution continues to work as expected. Also in test 4, since two followers failed, the solution is not impacted. Only the leader node cluster1 tries to connect to the two followers which are disconnected.

From the results of the tests, it is concluded that the solution is resilient for all four tests in this experiment.

Table 6.4: Experiments with Five Host Clusters

Cluster Failure							
	Test Case	**Cluster1**	**Cluster2**	**Cluster3**	**Cluster4**	**Cluster5**	**Is resilient?**
Test 1	Leader failed	X	✓	✓	✓	✓	Yes
Test 2	1 follower failed	✓	X	✓	✓	✓	Yes
Test 3	Leader and 1 follower failed	X	X	✓	✓	✓	Yes
Test 4	both followers failed	✓	X	X	✓	✓	Yes
Network Partition							
Test 5	Leader disconnected	✓	✓	✓	✓	✓	Yes
Test 6	1 Follower disconnected	✓	✓	✓	✓	✓	Yes
Test 7	Leader and 1 follower disconnected	✓	✓	✓	✓	✓	Yes
Test 8	2 followers disconnected	✓	✓	✓	✓	✓	Yes
Test 9	All clusters disconnected from each other	✓	✓	✓	✓	✓	No

.

Experiment 2: Network Partition

In this part, the POC was evaluated for network partition. A total number of five tests were performed to observe if the solution works as expected in the cases of network partition as shown in Table 6.4. Tests 5, 6, and 9 are the same as in section 6.1.2. However, tests 7 and 8 were added in this experiment. In test 7 the leader node and one follower node were disconnected from the three followers whereas in test 8 two followers were disconnected from the rest of the nodes.

The solution worked as expected in all 5 tests. In test 5, test 6, and test 9 the solution worked as in section 6.1.2. In test 5 and test 6, the single

partitioned node started reelection and waited to get the majority votes. Whereas the partition with the majority conducted reelection and selected a new node as the leader. In test 9, all the nodes kept trying to be elected as the leader until all of them were reconnected and one leader was selected.

In test 7, cluster1 and cluster2 were disconnected from cluster3, cluster4, and cluster5. In this case, since cluster1 lost the majority, leader election is started among cluster1 and cluster2. However, since the clusters fail to get the majority they kept trying and the election process kept getting timed out. Whereas on the other hand cluster3,cluster4 and cluster5 also started the leader election process but since they have the majority cluster4 was selected as the new leader and the solution worked on this part of the network partition.

Similarly in test 8 cluster2 and cluster3 get disconnected from cluster1, cluster4, and cluster5. In this case, also cluster2 and cluster3 started the leader election but kept failing. Whereas on the other hand since cluster1 already had the majority it kept functioning as usual.
From the results of the tests, it is concluded that the solution is resilient for all the tests included in this experiment except test 9 as expected.

Experiment 3: Time to Recover

In this experiment, the time that a follower KubeFed cluster takes to recognize the absence of a leader and take the leadership was observed when there were 5 host clusters. The experiment was repeated for five iterations for calculating the average recovery time of the POC. In the five host cluster scenario, when the leader cluster, i.e., cluster1 was taken down, the median value for the time it took for another cluster to observe that the leader is down is 2.702 seconds and the standard deviation is 0.554. Also, the median value for the time taken to start the leader election process and elect a new leader is 0.090 seconds and the standard deviation is 0.047. Overall the median value for the total time for the POC to recover from a failure is 2.790 seconds and the standard deviation is 0.527 as shown in Table 6.5

Table 6.5: Recovery Time for The POC with Five Host Clusters

	1	2	3	4	5	Median	Standard Deviation
Leader Failure Observed (s)	2.441	2.702	3.136	2.711	1.641	2.702	0.554
New Leader Elected (s)	0.056	0.090	0.003	0.120	0.110	0.090	0.047
Total Recovery Time (s)	2.497	2.790	3.139	2.831	1.751	2.790	0.527

.

6.2 Security

In this section, the security of the solution is evaluated by identifying the attack vectors that can be exploited by the attackers to compromise the security of the solution. In addition, this section also discusses the measures that can be taken to improve the security of the POC.

The next section describes four attack vectors in the POC.

6.2.1 Attack Vectors

Attack vectors are methods or pathways that are used by attackers to attack a network or a computer for exploiting a system. The POC designed in this thesis contains four attack vectors which are as follows.

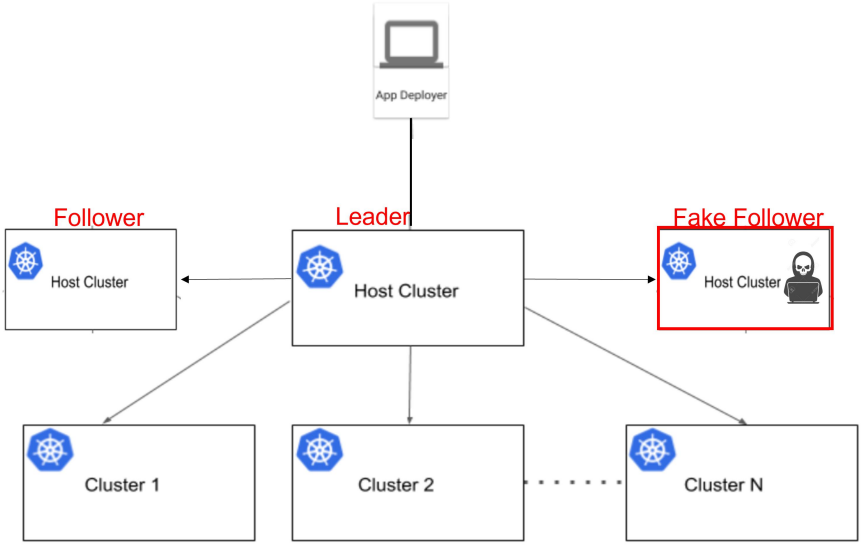

Figure 6.2: A Fake Follower Node Joined Raft

HTTP Server

In the POC, the first KubeFed host cluster that joins Raft is started in bootstrap mode and the other host clusters join the bootstrap node using

the HTTP address of the first host cluster and send their Raft address via an HTTP post request. The bootstrap node admits each cluster that sends a join request via HTTP post without authentication. If an attacker gets the information about the HTTP server, it can create a fake Raft node to join the cluster and send its Raft address as shown in Figure 6.2. The fake node would become part of the raft. It would get the information about the Raft address of all the nodes and would get information about all the Kubernetes resources deployed on the leader cluster compromising the confidentiality of the data. Also, if the fake node is selected as the leader, it can modify or delete the data by sending incorrect RPCs to the follower nodes impacting the integrity and availability of the data. The attacker can also impersonate the bootstrap node to prevent the followers from joining the actual HTTP node.

Raft Server

In the POC, the Raft nodes send their Raft address via HTTP post request while joining. The Raft server is used for sending two types of RPCs to the nodes, i.e., *RequestVotes* for requesting votes for leadership and *AppendEntries* for sending heartbeats and data for adding to the logs. If the attacker gets the information about the Raft server of any node, it can be exploited by the attacker in several ways. The attacker can impersonate a leader and send fake heartbeat RPCs to the followers when the leader fails so that there is no active leader. Moreover, the attacker can impersonate the followers and send fake confirmation to the leaders that the followers have committed the logs so that the leader commits the logs and whenever the leader would fail, the follower would fail to start from the same state due to the absence of logs. Furthermore, the attacker can send fake RPCs to the followers by impersonating the leader for creation, modification, and deletion of resources and when the follower would get the leadership, it might end up creating, modifying, or deleting incorrect resources compromising the integrity and availability of nodes. The attacker can also spoof the communication between the Raft nodes and read and tamper the confidential information about the resources as shown in Figure 6.3.

.

Bolt Store

The POC uses the bolt store to store the logs for the Raft nodes. The store is vulnerable to attacks from inside actors, i.e., people having physical or remote access to the KubeFed host clusters. The database can be accessed by anyone having access to the file system on the disk of the host

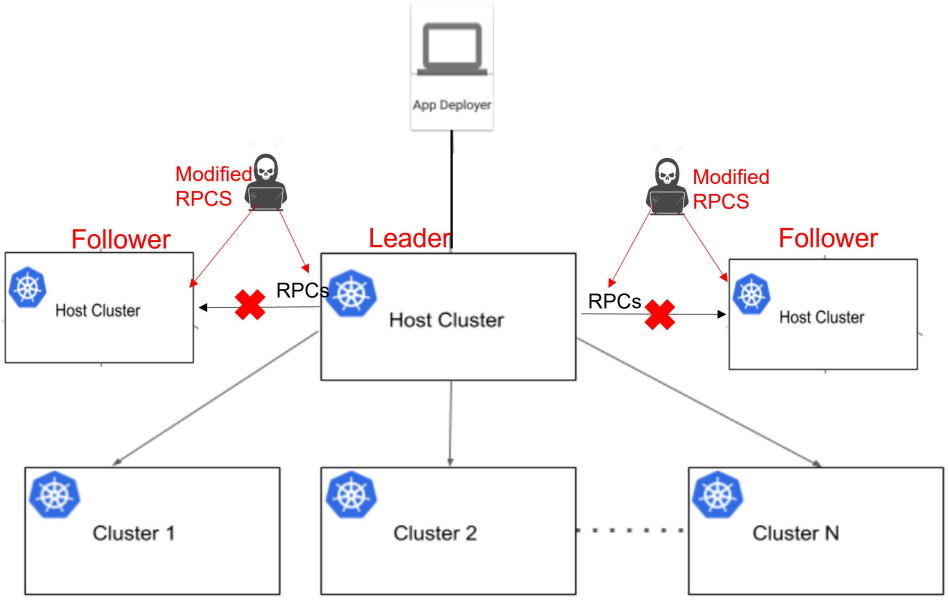

Figure 6.3: Attacker Tampering with RPCs

clusters. Moreover, the database does not authenticate the user before accessing the data. Anyone having access to any of the host clusters can access the database and read the data compromising the confidentiality of the data. Moreover, the attackers can create, modify and delete the key-value pairs storing the information about the federated resources compromising the integrity and availability of the system.

Hashicorp Raft Library Vulnerabilities

As Hashicorp Raft library is an open-source library there is a possibility of the presence of a number of vulnerabilities in the library as a result of usage of vulnerable methods. As the POC uses the Hashicorp Raft library, these vulnerabilities can be inherited or embedded to the POC from the Raft library. The library might also have some zero-day vulnerabilities that are not disclosed to the public yet. These vulnerabilities can be exploited by the attacker to compromise the security of the POC.

The next section describes the measures that can be taken to protect the POC from the attacks.

6.2.2 Security Measures

HTTP Server and Raft Server Security

For protecting the HTTP and Raft server from the attacker, two requirements need to be fulfilled. First, both the server and the clients for the HTTP and the Raft server need to be authenticated to prevent the attacker from impersonating the leader or the follower node. Second, the communication between the Raft nodes needs to be encrypted to prevent the attacker from spoofing and tampering with the data. To fulfill these requirements TLS [65] protocol can be used.

TLS is a cryptographic protocol that provides end-to-end security and prevents data from being eavesdropped or tampered with. It ensures that the communication between the two parties is secure. TLS provides three main features. First, it secures data from being eavesdropped on by encrypting the data transferred between two parties. Second, it prevents an attacker from impersonating a party by authenticating both parties communicating with each other. Third, it ensures that the data has not been forged or tampered with.

To prevent the attacker from impersonating the nodes and communicating with the HTTP and the Raft servers, a TLS authentication mechanism can be used to authenticate both parties involved in the communication. This is known as mutual authentication or two-way authentication in which both parties of the communication authenticate each other. TLS usually provides server-side authentication in which the client asks the server to prove its identity by providing a valid certificate. In mutual TLS, the server also asks the client to prove its identity by providing a certificate for mutual authentication between two parties.

Data encryption can be used to protect the data exchanged between the nodes and prevent the attacker from modifying or tampering with the data. TLS uses a combination of asymmetric and symmetric encryption. The server creates a public-private key pair. The client obtains the public key of the server and uses that key to encrypt a session key and sends that key to the server. The key is used by both parties during the session to encrypt the information.

Bolt Store Security

The data about the federated resources in the bolt store can be protected by implementing encryption at rest and authentication. The data can be encrypted on the disk to prevent the attacker from having an access to the

hard drive from accessing the database on the file system. Additionally, a user account specific to each cluster can be created that can be used by the POC in each cluster for accessing the database by logging in. The hash of the password of the user can additionally be used to encrypt the database that can be only decrypted when the user logs in using the password.

Hashicorp Raft Library Vulnerabilities Security

To protect the POC from the vulnerabilities present in the Hashicorp Raft library, it is important to be aware of the disclosed vulnerabilities in the library. These vulnerabilities are usually published on public platforms, such as mailing lists, commit-logs, change-logs, issue-trackers, and vulnerability databases including Common Vulnerabilities and Exposures (CVE). The platforms should be continuously monitored to be aware of the vulnerabilities and fix those in time.

6.3 Summary

In this chapter, the POC implemented in this thesis was evaluated for two scenarios. First, it was evaluated for reliability and high availability with three and five KubeFed host clusters. All the experiments performed for the reliability and high-availability tests worked as expected proving that the POC is highly reliable and resilient. Second, the POC solution was theoretically evaluated for its security by identifying the attack vectors and discussing the measures that can be taken to securing it against the listed attacks.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis by providing a brief overview of this thesis and listing the limitations of the proposed solution. In addition, this chapter lists the directions to move forward for improving the proposed solution.

7.1 Conclusion and Limitations

Kubernetes Clusters Federation provides a mechanism for managing application deployment on multiple clusters from a single management point known as the manager. This manager, also known as the host cluster in KubeFed is a single point of failure. If the manager or the host cluster fails, the application deployer cannot deploy applications on multiple clusters. Also, it is difficult to maintain the desired state of the member clusters. This thesis uses an active-passive high availability approach to solve this problem. Instead of using a single host cluster, multiple host clusters are used. Whenever the leader host cluster fails, one of the follower host clusters can transition to the leadership state and take its position.

In this thesis, a POC based on the proposed solution was developed. The POC passed all of the high-availability and reliability experiments presented in Chapter 6. According to the results of the tests, the proposed solution is feasible and can be used in practice. However, there are a few limitations of the current POC that can be addressed to improve its feasibility. The limitations are discussed below.

- **Size:** Raft works with a minimum number of 3 and a maximum number of 7 nodes with a recommendation of using 5 nodes for a production environment. To make Raft work on more than 7 nodes, sharding of the cluster is required where each shard runs a separate instance of

Raft. Therefore, the POC can only work for a minimum number of 3 and a maximum number of 7 KubeFed host clusters.

- **Hardcoded Addresses:** In each Raft node, there is an HTTP server used by the nodes to join the Raft cluster, the current POC uses hardcoded values of the HTTP server of the bootstrap node for sending a join request. This means that it is always required to configure the same node to be the bootstrap node. Also, the HTTP addresses of the nodes should be known in advance which is not always possible in production environments.

- **Transient Container Storage:** The current solution stores the Raft logs for the KubeFed host clusters in the container storage which is transient and is deleted whenever the container or the pod crashes. Therefore, whenever a container or the pod crashes the logs need to be transmitted from the current Raft leader causing additional overhead on the network.

- **Security Limitations:** The current solution has several attack vectors, such as the HTTP server, the Raft server, the bolt store, and the Raft library that can be exploited by the attackers to compromise the security of the solution as discussed in Section 6.2.1. For instance, an attacker can create a fake Raft node for joining Raft using the HTTP server and can get access to confidential information about the deployed resources. Similarly, the attacker can impersonate the leader or the follower node, send fake RPCs and compromise the integrity of the solution by modifying the resources.

7.2 Future Work

This section discusses the future directions for improving the proposed solution. The first idea is to explore the sharding of the KubeFed host clusters for deploying separate Raft instances. Also, more consensus algorithms that can support a large number of nodes can be explored. Although currently, there can be at most 7 nodes, in actual production workloads that are distributed across different geographical locations and thousands of clusters there might be a need to have more than 7 KubeFed host clusters. For this purpose, we can also explore and implement our own protocol for redundancy that can support more than 7 nodes.

Second, instead of using hardcoded values for the network addresses for the HTTP server of the nodes, a service discovery feature can be implemented

using a service mesh, such as Consul or Istio. With this feature, each node would register its HTTP address with the service that can be queried and used by the other nodes to join the server.

Third, instead of using transient container storage, a Kubernetes Persistent Volume [32] can be used. A Persistent Volume is a piece of storage in Kubernetes whose life cycle is independent of the container and the pod that uses it. This way the Raft logs would not be impacted if the container or the pod restarts.

Finally, to improve the security of the solution the measures discussed in section 6.2.2 can be implemented. For instance, TLS can be used for both the HTTP and the Raft server for server and client-side authentication and data encryption to prevent the attacker from accessing and modifying confidential information about the deployed resources.

There is also a production-ready Raft implementation rqlite [43] based on SQLite that provides several features, such as service discovery, authentication, and encryption using TLS. In the future, the feasibility of integrating rqlite with the current POC could also be evaluated.

It would be also beneficial to compare the approach proposed by this thesis to the one provided by Lars Lasrron et al. if they publish more details about their work in the future.

Bibliography

[1] Active-Active vs. Active-Passive High-Availability Clustering. https://www.jscape.com/blog/active-active-vs-active-passive-high-availability-cluster. Accessed 20.8.2021.

[2] Admiralty. https://github.com/admiraltyio/admiralty. Accessed 7.8.2021.

[3] Architecting Kubernetes Clusters. https://learnk8s.io/how-many-clusters. Accessed 7.8.2021.

[4] Bolt. https://github.com/boltdb/bolt. Accessed 20.8.2021.

[5] Ciluim Drawbacks. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed 7.5.2021.

[6] ClusterMesh. https://docs.cilium.io/en/v1.8/gettingstarted/clustermesh/. Accessed 20.8.2021.

[7] Consul. https://www.consul.io/docs/connect. Accessed 7.8.2021.

[8] Curl Command. https://curl.se/. Accessed 7.8.2021.

[9] Data Center Resiliency. https://searchdatacenter.techtarget.com/definition/resiliency. Accessed 20.8.2021.

[10] Hashicorp Raft. https://github.com/hashicorp/raft. Accessed 20.8.2021.

[11] Helm. https://helm.sh/. Accessed 20.8.2021.

[12] Is Just One Kubernetes Cluster Enough? https://blog.equinix.com/blog/2020/05/26/is-just-one-kubernetes-cluster-enough/. Accessed 7.8.2021.

[13] Istio Service Mesh. https://istio.io/. Accessed 7.8.2021.

[14] Kind. https://kind.sigs.k8s.io/docs/user/quick-start/. Accessed 20.8.2021.

[15] Kind Cluster Creation Script. https://github.com/kubernetes-sigs/kubefed/blob/master/scripts/create-clusters.sh. Accessed 20.8.2021.

[16] Kind Cluster Creation Tutorial. https://github.com/kubernetes-sigs/kubefed/blob/master/docs/environments/kind.md. Accessed 7.8.2021.

[17] KubeFed. https://github.com/kubernetes-sigs/kubefed. Accessed 7.8.2021.

[18] KubeFed Dynamic Client. https://github.com/kubernetes-sigs/kubefed/blob/2e8c13b6a5df2efdea39734bd42d7f2083419613/pkg/controller/util/resourceclient.go. Accessed 20.8.2021.

[19] KubeFed installation. https://github.com/kubernetes-sigs/kubefed/blob/master/charts/kubefed/README.md. Accessed 7.8.2021.

[20] KubeFed join members. https://github.com/kubernetes-sigs/kubefed/blob/master/docs/cluster-registration.md. Accessed 7.8.2021.

[21] Kubefedctl. https://pkg.go.dev/sigs.k8s.io/kubefed/pkg/kubefedctl. Accessed 7.8.2021.

[22] Kubernetes API. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. Accessed 7.8.2021.

[23] Kubernetes ConfigMap. https://kubernetes.io/docs/concepts/configuration/configmap/. Accessed 7.8.2021.

[24] Kubernetes Custom Resource Definitions. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. Accessed 20.8.2021.

[25] Kubernetes Deployment. https://www.vmware.com/topics/glossary/content/kubernetes-deployment. Accessed 7.5.2021.

[26] Kubernetes Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Accessed 7.5.2021.

[27] Kubernetes Federation Evolution. https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/. Accessed 7.8.2021.

[28] Kubernetes Namespace. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. Accessed 7.8.2021.

[29] Kubernetes Namespace. https://www.vmware.com/topics/glossary/content/kubernetes-namespace. Accessed 7.8.2021.

[30] Kubernetes Network Limitations. https://zhaohuabing.com/post/2020-02-21-network-service-mesh-english/. Accessed 7.8.2021.

[31] Kubernetes Objects. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. Accessed 7.8.2021.

[32] Kubernetes Persistent Volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Accessed 7.8.2021.

[33] Kubernetes Pods. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed 7.5.2021.

[34] Kubernetes Secrets. https://kubernetes.io/docs/concepts/configuration/secret/. Accessed 20.8.2021.

[35] Kubernetes Service. https://kubernetes.io/docs/concepts/services-networking/service/. Accessed 7.8.2021.

[36] Kubernetes Tools. https://kubernetes.io/docs/tasks/tools/. Accessed 7.8.2021.

[37] Multi-cloud, Multi-region Kubernetes federation - Part 1. https://laptrinhx.com/multi-cloud-multi-region-kubernetes-federation-part-1-3444634735/. Accessed 20.8.2021.

[38] MySQL. https://www.mysql.com/. Accessed 20.8.2021.

[39] Network Service Mesh. https://networkservicemesh.io/. Accessed 7.8.2021.

[40] PostgreSQL. https://www.postgresql.org/. Accessed 20.8.2021.

[41] RBAC. `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`. Accessed 20.8.2021.

[42] Resiliency vs Resundancy. `https://www.bmc.com/blogs/resiliency-vs-redundancy/`. Accessed 20.8.2021.

[43] rqlite. `https://github.com/rqlite/rqlite`. Accessed 7.8.2021.

[44] Shipper. `https://github.com/bookingcom/shipper`. Accessed 7.8.2021.

[45] Submariner. `https://submariner.io/`. Accessed 20.8.2021.

[46] Understanding Multi-Cluster Kubernetes. `https://www.getambassador.io/learn/multi-cluster-kubernetes/`. Accessed 7.8.2021.

[47] VirtualBox. `https://www.virtualbox.org/`. Accessed 20.8.2021.

[48] Wget. `https://www.gnu.org/software/wget/`. Accessed 7.8.2021.

[49] Kubernetes Components, March, 2021. `https://kubernetes.io/docs/concepts/overview/components/`. Accessed 7.8.2021.

[50] Kubernetes Home, Nov, 2020. `https://kubernetes.io/docs/home/`. Accessed 7.8.2021.

[51] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue 14*, 1 (Jan. 2016), 70â93.

[52] Faticanti, F., Santoro, D., Cretti, S., and Siracusa, D. An application of kubernetes cluster federation in fog computing. In *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)* (2021), pp. 89–91.

[53] Howard, H., and Mortier, R. Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (New York, NY, USA, 2020), PaPoC '20, Association for Computing Machinery.

[54] Huang, J., Xiao, C., and Wu, W. Rlsk: A job scheduler for federated kubernetes clusters based on reinforcement learning. In *2020 IEEE International Conference on Cloud Engineering (IC2E)* (2020), pp. 116–123.

[55] Huaxin, S., Gu, X., Ping, K., and Hongyu, H. An improved kubernetes scheduling algorithm for deep learning platform. In *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)* (2020), pp. 113–116.

[56] Jain, N., and Choudhary, S. Overview of virtualization in cloud computing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)* (2016), pp. 1–4.

[57] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)* (2011), pp. 245–256.

[58] Kim, D., Muhammad, H., Kim, E., Helal, S., and Lee, C. Tosca-based and federation-aware cloud orchestration for kubernetes container platform. *Applied Sciences 9* (01 2019), 191.

[59] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133â169.

[60] Lamport, L. Paxos made simple.

[61] Larsson, L., Gustafsson, H., Klein, C., and Elmroth, E. Decentralized kubernetes federation control plane. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)* (2020), pp. 354–359.

[62] Medel, V., Tolosana-Calasanz, R., Ãngel Banares, J., Arronategui, U., and Rana, O. F. Characterising resource management performance in kubernetes. *Computers Electrical Engineering 68* (2018), 286–297.

[63] Nguyen, N. D., and Kim, T. Balanced leader distribution algorithm in kubernetes clusters. *Sensors 21*, 3 (2021).

[64] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[65] Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018.

[66] Rullo, A., Serra, E., and Lobo, J. *Redundancy as a Measure of Fault-Tolerance for the Internet of Things: A Review*. Springer International Publishing, Cham, 2019, pp. 202–226.

[67] Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaisse, B., and Joosen, W. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware* (New York, NY, USA, 2016), ARM 2016, Association for Computing Machinery.

[68] Van Renesse, R., and Altinbuken, D. Paxos made moderately complex. *ACM Comput. Surv. 47*, 3 (Feb. 2015).

[69] Vayghan, L., Saied, M., Toeroe, M., and Khendek, F. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. pp. 176–185.

[70] Vohra, D. *Kubernetes Management Design Patterns: With Docker, CoreOS Linux, and Other Platforms*, 1st ed. Apress, USA, 2017.

[71] Xing, Y., and Zhan, Y. Virtualization and cloud computing. In *Future Wireless Networks and Information Systems* (Berlin, Heidelberg, 2012), Y. Zhang, Ed., Springer Berlin Heidelberg, pp. 305–312.

Appendix A

First appendix

This section presents the details for setting up the test environment by providing detailed configuration steps for cluster deployment, KubeFed installation, POC installation, and RBAC configuration. The setup requires Go 1.15 to execute.

A.1 Cluster Deployment

To create a Kind cluster, the user can run the following command on Kind version `v0.10.0`.

```
kind create cluster --name CLUSTER-NAME IMAGE
```

The command would create a kind cluster with the specified name and image and the default configuration that includes a single control plane node and no worker nodes. However, the default configuration can be modified to customize the number of control plane as well as worker nodes. The cluster can be accessed using the configuration that is created and stored by Kind at *{HOME}/.kube/config*. The configuration file describes clusters, users, and contexts. The context groups access parameters to a cluster including cluster name, namespace, and the user as shown in Figure A.1.

In order to keep the experiments simple, the default configuration has been used and the workloads are executed on the control plane node. To create the required number of Kind clusters, a shell script [15] from the KubeFed GitHub repository has been used. The desired number of clusters can be provided to the script by modifying the value of `NUM_CLUSTERS` variable in the script as shown in the tutorial [16]. The script would create the specified

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://172.18.0.2:6443
  name: kind-cluster1
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://172.18.0.3:6443
  name: kind-cluster2
contexts:
- context:
    cluster: kind-cluster1
    user: kind-cluster1
  name: cluster1
- context:
    cluster: kind-cluster2
    user: kind-cluster2
  name: cluster2
current-context: cluster1
kind: Config
preferences: {}
users:
- name: kind-cluster1
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
- name: kind-cluster2
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

Figure A.1: Cluster Configuration File

number of Kind clusters and change the context name of the cluster to the cluster name. The current context is the default cluster that can be accessed using kubectl. The user can change the current context to any of the clusters by using the command.

```
kubectl config use-context CONTEXT-NAME
```

The next section describes the KubeFed installation process on the system.

A.2 KubeFed installation

KubeFed can be installed on the designated host clusters using Helm [11] as presented in the tutorial [19]. Helm is a package manager used in Kubernetes to describe application structure through helm-charts and manage the structure using simple commands. It lets the users create a collection of application resources and deploy the application as a single unit. Helm can be used by the users for installing, managing, and updating even the most complicated Kubernetes applications. Using Helm version v3.5.3, first, the KubeFed chart version 0.7.0 is added into the local charts by using the following command.

```
helm repo add kubefed-charts https://raw.
    githubusercontent.com/kubernetes-sigs/kubefed/
    master/charts
```

Afterwards, KubeFed is installed on all the host clusters using the command.

```
helm --namespace kube-federation-system upgrade -i
    kubefed kubefed-charts/kubefed --version=0.7.0
      --create-namespace
```

The member clusters can join the host clusters using the following kubfedectl [21] command. Kubefedctl is a command-line utility for KubeFed.

```
kubefedctl join MEMBER-CLUSTER --cluster-context
    MEMBER-CLUSTER --host-cluster-context HOST-
    CLUSTER-CONTEXT --v=2
```

In the above example, MEMBER-CLUSTER is the name of the cluster that joins the host cluster and HOST-CLUSTER-CONTEXT is the name of the context of the host cluster that the member cluster intends to join.

The detailed steps for joining and unjoining the host clusters are presented in the tutorial [20]

The next section describes the steps for installing the POC on the KubeFed host clusters.

A.3 POC Installation

The POC is packaged as a docker image and uploaded onto the docker hub in a private registry. The image is then installed on the KubeFed host clusters in a pod as a Kubernetes deployment. The configuration file of the deployment is shown in Figure A.2. The deployment is exposed outside the cluster by setting the `hostNetwork` to true.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: raft-deployment
  labels:
    app: raft
spec:
  replicas: 1
  selector:
    matchLabels:
      app: raft
  template:
    metadata:
      labels:
        app: raft
    spec:
      hostNetwork: true
      containers:
      - name: raft-app
        image: IMAGE-NAME
        imagePullPolicy: Always
        ports:
        - containerPort: 30001
```

Figure A.2: POC Deployment Configuration

The following command can be used for deploying the POC on the des-

ignated KubeFed host clusters.

```
kubectl apply -f DEPLOYMENT_FILE
```

The next section describes the RBACs created for the POC.

A.4 RBACs

RBACs contain four types of objects i.e., *Role*, *ClusterRole*, *RoleBinding*, and *ClusterRoleBinding*. This thesis uses two of these objects which are described below:

- **ClusterRole:** contains rules representing a set of additive permissions. It can be used to define permissions on cluster-scoped objects.
- **ClusterRoleBinding:** is used to apply the set of rules defined in the ClusterRole to cluster-wide objects.

Figure A.3 shows an example of the configuration of a ClusterRole created for watching federated namespace.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: fns-reader
rules:
- apiGroups: ["types.kubefed.io"] # "" indicates the
     core API group
  resources: ["federatednamespaces"]
  verbs: ["get", "watch", "list", "delete", "create
      ", "update"]
```

Figure A.3: ClusterRole for Federated Namespace

The above ClusterRole can be applied to the clusters using Cluster-RoleBinding on the default account using the following command.

```
kubectl create clusterrolebinding fns-reader --
    clusterrole=fns-reader --serviceaccount=default:
    default
```